A Programming Language and Compiler
Based on an Augmentation of
the ACM Model

by

Thomas Edgar Shirley

B. S., Carnegie-Mellon University, 1980

A MASTER'S THESIS

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1987

Approved by:

Major Professor

CONTENTS

LIST OF FIGURES

Acknowledgements

The completion of this thesis is the result of a team effort and I'd like to take this opportunity to thank my team. Bob Fish and Dick Yuknavech were my partners in this project, as well as in a few others in the past three years. Their inspiration, leadership, and brain power have been indispensable in my intellectual growth. I hope we can continue our teamwork until Dick finally retires to Hawaii in 1994. Dr. Unger not only provided the dissertation which serves as the basis for this thesis, but also provided guidance, leadership, open mindedness, and friendship throughout the process. I'd also like to thank the committee, Dr. Wallentine and Dr. Schmidt, for their time and interest in this work.

As anyone who has undertaken a thesis understands, emotional stability is very difficult to maintain. I'd like to thank my parents, Nora, and Dr. Jim Langlas who all helped in their own special ways.

CHAPTER 1

1. Introduction

1.1 Problem Overview

Over the past decade multiple processor architectures have moved from the realm of the research laboratory towards feasible and marketable products. Many factors have contributed to this shift from uniprocessor to multiple processor computing among which are:

- Cost per cycle is significantly better for microprocessors than mainframes,
- Realization of the Von Neumann bottleneck [Bac78] and its limitations on a uniprocessor's performance capabilities,
- VLSI Technology has improved size, speed and reliability of components [Gaj83],
- Physical limitations (such as the speed of light) are being approached limiting future hardware improvements in uniprocessor performance, and
- Networking technology has improved to the point where it is feasible and practical to link machines together [Gaj83].

In theory the raw performance advantage of a uniprocessor may be exceeded by a collection of smaller machines. This synergy can be provided if the computing tasks are effectively partitioned and dispatched throughout the processors. The ideal partitioning insures independence of each partition. The ideal dispatching guarantees that all of the available processors are kept productively busy processing partitions.

Two major obstacles bar displacement of the uniprocessor in the marketplace:

- Accommodation of the existing software base. Business simply cannot afford the costs associated with redeveloping or purchasing software for the new environment and the subsequent trouble reporting and correction intervals. Compatibility standards, transformational tools, and virtual machines are techniques to adapt existing software into the new environment.

- Software development tools appropriate for the concurrent environment. Programming and job control languages, and debuggers are the most sensitive to this environment. Additionally, the burden of interacting with the complexity of the network should be handled by the tools and operating systems. Positioning this complexity management within the systems software yields an economy of scale improvement for each user and as such will increase productivity. Application programmers can concentrate on solving the application task at hand rather than continually resolving the environmental problems.

1.2 Related Research

Extensive research has been carried out within the hardware community on parallel processing. Architectures have been proposed and constructed which can be categorized as [Fly79]:

SISD Single Instruction Single Data. This is the conventional Von Neumann architecture of a CPU and memory connected through a memory access path. Parallelism has been introduced into this architecture through overlapped CPU operation and memory access. However the Von Neumann bottleneck of the memory access path bounds performance.

SIMD Single Instruction Multiple Data. Array and vector processors fall into this architectural classification. This approach executes a single instruction on many data

items simultaneously. Mathematical applications which perform matrix transformations are particularly amenable to this approach. Opponents of this approach argue that the parallelism gained is limited to a small set of problem domains [Alm85].

MISD Multiple Instruction Single Data. The execution of numerous instructions on a single data item characterize a pipeline strategy. A particularly attractive application of pipelining is the execution of floating point operations [Ens74]. A disadvantage of this strategy is the long delay associated with pipeline startup.

MIMD Multiple Instruction Multiple Data. Multiple processors connected through an interconnection mechanism are grouped in this category. The processors function independently with a minimum amount of inter-processor synchronization. It has been argued that the interconnection network becomes the bottleneck in this arrangement. Experience has shown [Jon80] that acceptable levels of performance can be achieved.

Hardware research has made considerable strides and has actually constructed many different multiple processor systems ([Alm85] and [Gaj85] survey the field). Software research has lagged behind; in many cases driven principally by the need for software to run on the hardware architecture at hand. The construction of software for specific machines can be argued to be the most efficient in terms of resultant execution. From a practical standpoint this re-invention of the software wheel is time consuming, expensive, and inherently non-portable [Pad80]. Additionally, approaches which leave the machine visible to the programmer force thinking along these lines rather than along the lines of the solution itself [Ack79].

Partitioning and scheduling are the two major software problems to be solved in the multiple processor environment. The problem must be partitioned into independent pieces and these

partitions subsequently scheduled for execution as processors are available. A simple example will illuminate the difficulties. Figure 1-1 gives three simple statements and the initial values for the variables used in these statements. Three of the variables, a, d and f are undefined prior to execution. Under sequential execution the results are correctly computed as $a=5$, $d=5$, and $f=18$.

A possible partitioning strategy is to group statements so that data access is disjoint between partitions. Correct results for this example can only be obtained by grouping all three statements into the same partition and executing this partition sequentially. For example, partitioning the statements as in figure 1-2 and assuming that the partitions may execute in any order may lead to partition P3's evaluation prior to completion of partition P1 or P2. For this example partitioning alone will not improve upon sequential execution.

```
S1:    a ← b + c
S2:    d ← e
S3:    f ← a + d + b + e
```

Initial values:
a = ?
b = 3
c = 2
d = ?
e = 5
f = ?

Figure 1-1. A Simple Partitioning and Scheduling Problem

An alternative is to weaken the condition that the partitions be independent and enforce an ordering on the execution. In this case figure 1-2 is a valid partitioning and the results will be correct if partitions P1 and P2 are arranged to be completed prior to the initiation of partition P3. Partitions P1 and P2 may be executed simultaneously.

```
S1:    a ← b + c              P1
-----------------------------------
S2:    d ← e                  P2
-----------------------------------
S3:    f ← a + d + b + e      P3
```

Figure 1-2. A Partitioning of Figure 1-1

This example is simple enough that the dependencies which dictate the partitioning and scheduling may be intuitively seen. Real world problems require formal and robust methods to discover the correct partitioning and scheduling. Research is divided into explicit and implicit methods.

In an explicit partitioning of a program the programmer indicates the portions of the code which may be executed concurrently. Explicit concurrency has been achieved by augmenting sequential languages with concurrent constructs and by inventing new languages embodying these constructs. The Ada task, PL/I entry, and Concurrent Pascal process are a few such constructs. [And83], [Geh84], [Ghe85], and [Wil81] compare and contrast programming languages offering explicit concurrency.

A major problem with explicit concurrency is that the robustness of programs is often a function of the programmer's ability. Incorrect specification can lead to non-functional behavior [Han73]. An additional problem is that programmers may overlook opportunities for concurrency and therefore not achieve maximum performance.

Implicit partitioning on the other hand requires inspection of the program to discover potential concurrency. This technique is performed through data flow analysis. Analysis consists of constructing a directed graph in which the nodes represent instructions and the arcs represent

data paths. Tokens representing data values pass along the graph. A node is eligible for execution when tokens are available on all incoming data paths. This execution scheme is referred to as data drive. Dataflow closely resembles Petri Net theory [Pet77].

Figure 1-3 gives the dataflow graph for figure 1-1. The initial values are given as sources, the results as sinks. This graph clearly depicts that statements S1 and S2 are independent and statement S3 is dependent upon both.

Using a dataflow graph to determine partitioning and ordering is very natural. [Dav82] discusses this technique based upon two types of dataflow operation: token and structure. [Ack79] surveys three dataflow programming languages, VAL, ID, and LAU. The SISAL dataflow language is described in [Gur85].

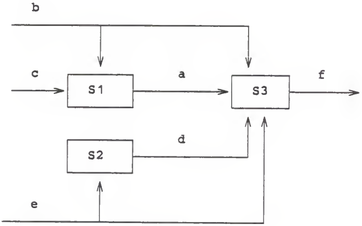


Figure 1-3. Dataflow Graph for Figure 1-1

The Parafrase compiler project [Gaj83] takes both implicit and explicit approaches towards concurrency. The programmer codes with sequential Fortran and presents this to Parafrase. The compiler performs a series of algorithms which expose parallelism ([Pad80], [Dav81], [Cyt82], and [Har85]). The compiler's output is an augmented Fortran program which the programmer

may wish to maintain (at which point parallelism is explicitly controlled by the programmer). Studies have shown that this technique yields sufficient parallelism to drive a multiprocessor system [Kuc74].

The concept of granularity distinguishes systems which follow the dataflow paradigm. In a finely granular system the processing elements work at a very low level, for example the machine instruction level. The theory underlying fine granularity is that this microscopic view of the problem will produce massive parallelism which is not available at a higher level. Fine granularity is used in Dennis' dataflow machine [Den80], the tagged token machine of Arvind [Arv80], and the Manchester dataflow machine [Gur85].

At the other end of the spectrum is coarse granularity. Course granularity assigns processing elements groups of instructions. Massive amounts of parallelism are sacrificed in favor of minimized synchronization and scheduling. This tradeoff is advantageous when dealing with regular forms of data such as arrays [Gaj85] or when synchronization and/or scheduling is expensive as in a network of microprocessors.

A different approach to partitioning is the functional style. This approach is based on the strict mathematical notion of a function. Since the mappings are strict, with no side effects, functions may be evaluated at any time. This, in effect, is implicit concurrency. Backus [Bac78] motivates the need for this approach, describes the benefits and his FP system. A survey of current functional programming is contained in [Kog85].

Concurrency in conventional languages has often been troubled by the presence of side effects. The problem pertains directly to the accessibility and volatility of global data. This problem has been addressed in three different ways; definitional, message passing, and shared memory synchronization.

The definitional approach to side effects is used by functional languages and the dataflow languages. Functional languages, by definition, do not contain side effects. Dataflow languages have adopted the single assignment rule [Com78]. This rule permits only a single binding of a value to an object (variable). Side effects are eliminated because the object's value can never change once it is bound. This is particularly amenable to dataflow in which computation is driven by data availability.

The message passing approach is a derivative of many years of productive research on information hiding [Par72] and data abstraction [Lis77] and [Wul76]. This approach prohibits global manipulation of shared data (as such) and instead requires access through a set of defined operations. The representation of the object is known only within the encapsulation (cluster) and only manipulated within the defined operations. The primary implementation of this approach is to discipline all data exchange through message passing. CSP [Hoa78], PLITS [Fel79], CLU [Lis77] and Alphard [Wul76] are a few languages which obey this methodology. A more detailed comparison of these languages, and a description of others, is contained in [And83], [Wil84], and [Ghe85].

In the shared memory approach there exists the potential for side effects. As such correct access to global data (shared data) requires disciplined synchronization. Semaphores [Dij65] permit a very microscopic approach to synchronization. Painful experience has proven that this is a very difficult construct to master and has led to the higher level mechanisms of conditional critical regions [Hoa72] and monitors [Hoa74]. Detailed comparisons of these techniques and programming languages which support them are also contained in [And83], [Wil84], and [Ghe85].

1.3 Overview of Thesis

This thesis is part of an investigation into building a programming language, compiler and a supporting operating system based on the ACM computational model [Ung78a] and [Ung78b]. The model supports implicitly concurrent computation as identified by data drive. The model is flexible enough to handle both fine and course granularity and abstract enough to divorce itself from any particular processor architecture. The goals of this research have been to realize the model in a programming language, identify and solve shortcomings of the model, and develop an operational system with which to gain further experience.

This thesis begins by examining the model in depth by attempting to solve numerous problems. These problems will in some cases be easily solved and in others illuminate weaknesses of the model. The problem solving exercise will present the need for two augmentations to the model; an indefinite looping construct and a pipelined treatment of dynamic data objects. These additions to the model will be motivated, defined, and demonstrated in chapter three. The two additions when used together allow modeling of many complex, real world problems. One such problem will be presented at the conclusion of chapter three.

Implementation of the model in a programming language is examined in terms of the language's compiler. The syntax for the language is given as well as the symbol table structure. The compiler's output is generated for a virtual machine. The virtual machine's characteristics and execution model are described.

The fifth and final chapter will elaborate the contributions of this thesis and identify issues which were encountered and left unsolved by this study.

CHAPTER 2

2. Examination of the ACM Model

This chapter introduces and summarizes the ACM computational model [Ung78a] and [Ung78b]. This model is the basis for this thesis. In an effort to gain experience with the model, the second section will present some classical and typical problems and attempts to solve these problems within the model. The model will easily solve some of the problems. In other cases, deficiencies with the model, or the unnaturalness of expression, will lead to postulation of extensions or refinements. Some of these weaknesses will be remedied within the third chapter of the thesis. Others will be left for further research.

2.1 Summary of the ACM Model

ACM is an intrinsically concurrent computational model. The model is based on the notion of an object which encompasses both data and action. Data drive is the primary mechanism for ordering the execution of actions. Closer control may be obtained by augmenting actions with stimulation and termination predicates. Constructs for alternation, caseation, iteration, repetition, and recursion are defined to increase the expressive power of the model.

Data and procedural abstraction is utilized by the model. Data abstraction is provided by aggregation. The detailing concept provides procedural abstraction. The presence of abstraction in the model allows support of a wide range of applications and underlying architectures. For example, the base level actions could range in granularity from machine instructions, to procedures, to entire jobs. A primitive data type could actually be at the machine representation, a message structure, or even a file.

A kernel for the model has been characterized in terms of allowable state transitions of objects. This characterization includes necessary and sufficient conditions for deadlock. A subset of the kernel has been further examined and proven to be determinate.

2.1.1 Object Basics An object is the fundamental building block in the ACM model. Both data and action are encapsulated in the object definition. An object is defined to be the quintuple (d, a, r, c, v) corresponding to designator, attribute, representation, corporality and value respectively. An object is defined to exist when its designator exists. The value of an object, which is the basis of data drive, exists when all components of an object exist.

The *designator* component of an object is defined as the quadruple (c, u, i, a) where c is the context, u the user defined name, i the instance, and a the alias name.

The *context* represents the hierarchy of the object's existence. When an object is created, it is given the current environment as its context. Context is augmented as an object is passed up to higher levels within the environmental hierarchy.

The *user defined name* is an arbitrary list of names specified by the user. *Alias* is a collection of alternative names which may be used to access this designator.

The *instance* component is a triple $(s, (t, tc), (o, oc))$ which differentiates a designator with identical context and user name. s is a spatial position, t is a chronological identity which is generated through a user defined clock function tc, and o is a sequence number with initial value oc.

An important quality of the model is the ability to access, in a well defined manner, an object (or objects) by specifying only a portion of the designator(s). This is an alternative to aliasing. The approach is to match on all the designator components specified. If this does not yield an

exact match then hueristics are applied in an attempt to find matches. The hueristics used are:

- Broaden the contextual hierarchy if a context match is not made,
- Obtain a collection of user names which are more refined then the one specified,
- Gather all unspecified dimensions, and
- Use the most recent chronological or sequential incarnation.

Attribute is the logical notion of an object's type, internal structure, and relationship to other objects. The type may be atomic (boolean, integer, real or character) or a structure. Four special structure types are included as part of the model: ordered and unordered collections, sets, and actions. The internal structure refers to the relationship of components of a structure. The relationship to external objects may also be expressed.

Structures are created through the aggregation of one or more objects (which may, recursively, be structures themselves). A structure is defined as the triple (o, p, c) where o is the objects comprising the structure, p defines partitions which the structure may then take on, and c defines the legal operations on the structure. This definition allows a structure to act as an abstract data type.

Representation is the physical notion of an object. This includes location, coding scheme and packing considerations.

Corporality is the quadruple (l, p, r, e). The p component refers to the *place* where the object may be located. e gives the *authorization* required to use the object.

l is the *longevity* of the object. Longevity categorizes the binding time and strength of a value to the designator. Four longevities are defined: fixed, static, dynamic, and fluid. Objects with

fixed longevity have their designators and values bound at the inception of the model. *Static longevity* objects are bound once by actions within the model. Objects with static longevity obey the single assignment rule. *Dynamic longevity* objects are also bound within the model. Incarnations of the designators are maintained. To the user of a specific dynamic object the object obeys the single assignment rule. However, relative references to dynamic objects allow the objects to appear to change over time. The final category is fluid. *Fluid longevity* objects may change over time with no historical record maintained.

Replication of an object is defined by the r component. Replication is defined as the triple (a, s, g) where a represents the object's availability, s represents the number of identical copies of the object, and g is a boolean indicating whether the object may be copied.

The final component of an object is *value*. Values are of the type specified in the attribute component. The action type and value is the topic of the next section.

To aid in the understanding of the definitions just presented an example is in order. Consider the definition of an object which represents a master's thesis.

The designator of the object consists of a context, user name, and instance. The graduate student's home represents the initial context of creation. Progressively more general contexts which the thesis may enter are the major professor's office, department file, and university library. The user name will be the graduate student's last name. Some refinements on the name are allowable such as the title of the thesis, defense date, and publication date. Spatial position is used to represent the thesis as a vector. Each component of the vector references a particular chapter. Chronological identity refers to the dates of successive revision and sequencing is done for each of these revisions.

Figure 2-1 gives some references to the thesis object and explains what will be obtained for each reference. Assume that the student's name is Jones and the major professor is Smith, a computer science professor. Jones lives on Elm Street and Smith's office is in Room 100.

Reference	Object(s) obtained
Jones	The latest version of Jones' entire thesis.
Elm.Jones	Another reference to Jones' entire thesis.
CS.Room100	All theses produced in the computer science department supervised by Smith.
Jones(2)	Most recent version of the second chapter of Jones' thesis.
Jones(2)..0	The first version of chapter 2.
Jones..(2)+0	The latest version of chapter 2.
Jones.title	The title of Jones' thesis.
Jones.yesterday	Yesterday's version of Jones' thesis.

Figure 2-1. Object References and Values Obtained

The thesis object would be defined to be a dynamic object because it changes over time and it is advantageous to maintain versions over and above the most recent. The object can be replicated to allow simultaneous review and edit, however edit authorization is given only to the graduate student. The representation of the object is best done in terms of an abstract data type. Three operations are allowed: edit, read, and print.

2.1.2 Action Related Definitions The previous section introduced the object concept. The example, and much of the discussion, centered around the data aspects of an object. A non-intuitive variant of the object is the action.

Actions for which the value component is a computational device within the modeled (or physical) environment are primitives. The model is flexible enough to permit these to be hardware devices, machine instructions, or coarser grained such as a process or a job. Actions may also be defined recursively; an action's value may consist of additional requests for actions.

A *simple action* is a triple (m, a, r) where m is a list of the materials used by action a to compute the results listed in r. A *simple request* is identically defined. This is confusing, however recall that an action is an object. The request is a reference to this object. In terms of imperative languages, the action is a function specification with formal parameters, the request is a function invocation with actual parameters.

As noted previously data drive is the primary method for execution ordering. This can be augmented by the introduction of conditions. A *stimulation condition* is a boolean expression which must be true, in addition to data drive, in order for a request to be marked eligible for execution. A *termination condition*, when true, terminates a running request or marks a request as ineligible for subsequent execution.

Simple requests and actions are augmented to include conditions. The conditions placed on actions are called *internal stimulation* and *internal termination*. Those placed on requests are called *external stimulation* and *external termination*. Thus requests and actions are defined by the following quintuple:

$$\text{Action} ::= (s_i, m, a, r, t_i)$$

$$\text{Request} ::= (s_e, m, a, r, t_e)$$

Figure 2-2. Action and Request Definitions

Stimulations and terminations place additional constraints upon execution ordering. *Partialing*

relaxes the data drive requirement by allowing the execution of a request prior to the existence of indicated materials. This concept is very useful for objects which are seldom needed, or for requests which entail a substantial startup period prior to the partialed materials' usage.

The process of elaborating an action is termed detailing. The requests which make up the detail for an action are termed its *request set*. Side effects are avoided in detailing by requiring material lists used within the request set to be derived from the material list of the action or from result lists of other requests in the same request set. On the other hand, results in the action's result list must be produced by some member of the request set. An important consequence of this is that if a request terminates successfully, it produces all of its results. If a request terminates unsuccessfully none of its results are produced.

2.1.3 Constructs Constructs are defined in the model to provide alternation, caseation, repetition, iteration, and recursion. These provide the control flow capabilities necessary for modeling with ACM. A description of the notation used in examples in this section is given and then each of the constructs is treated in turn.

A request is denotated by:

$$[\, s_e] \;:\; n\,(\, m\;; r\,[t_i])\;\; [t_e\,]$$

where:

s_e	-	External Stimulation condition
n	-	Name of the request
m	-	Material list to the request
r	-	Result list from the request
t_i	-	Internal Termination condition
t_e	-	External Termination condition

Figure 2-3. Request Notation

Alternation is akin to the imperative if-then-else construct. It is achieved by augmenting two requests with mutually exclusive external stimulations. The external stimulations must be

complements of each other to provide for all possibilities. Nested decisions may be achieved by detailing or through the caseation construct.

A simple example of alternation is to find the square root of a given number N. If the number is positive use algorithm A and if negative use B to accommodate imaginary numbers.

```
[N >= 0]   :  A(N; SquareRoot)
[N <  0]   :  B(N; SquareRoot)
```

Caseation allows an arbitrary number of requests to be augmented with mutually exclusive external stimulations. The caseation stimulations need not be universal. A simple example of caseation which occurs in a database management system would select a request based on the transaction indicator. Any invalid transaction indicators are ignored.

```
[transaction = 'add']   :  Add(tuple; status)
[transaction = 'del']   :  Delete(tuple; status)
[transaction = 'upd']   :  Update(tuple; status)
```

Repetition allows the notational abbreviation of a group of requests in which the materials and requests vary in a well behaved manner. For example, double each element of a ten element vector V to produce a new vector W could be expressed as:

```
Mult(V(1), 2; W(1))
Mult(V(2), 2; W(2))
    .
    .
Mult(V(10), 2; W(10))
```

or with repetition as:

```
Mult(V, 2; W)
```

The *iteration* construct also allows the notational abbreviation of repeated requests. Iteration is used when definite or indefinite sequences of the same request are to occur. One material element of the material list of the request is allowed to vary with each iteration. This material is

the result of a previous request in the iteration (or the result of an initial setup). This relationship dictates a rigid sequence. The iteration is terminated through an internal termination which is a function of the changing result. A binary search through an ordered vector V for an element E could be carried out as:

BinarySearch(V, I; I [I != -1 and V(I) != E])

Recursion is attained by allowing requests in the request set of an action to name the action. Material and results are constrained to be local incarnations with binding of result list objects deferred until an action's termination. The effect of local constraint is to mimic the stack approach to recursion of imperative languages. Binding of results at action termination allows dynamic objects to be created as the stack unwinds.

2.1.4 Kernel The model characterizes a kernel for ACM with a few fundamental states and conditions for transitions between these states. An object can be non-existent, existent but unavailable, or existent and available.

A request can take on the states: idle, enabled or disabled. A request is made idle when its containing action is enabled. A request may subsequently transition to the enabled state subject to data drive and stimulation conditions. Once enabled an iteration may return to idled if upon completion its internal termination is false. An enabled request becomes disabled if it runs to completion. A condition in either the idled or enabled state is disabled if its termination condition evaluates to true.

Two exceptional situations may occur which effectively block an action's completion: hangup and deadlock. A *hangup* state is attained when all requests within the detail are disabled or idled. A *deadlock* situation may occur when a circular dependency exists between materials and

results of two or more requests which detail an action. Deadlock occurs when all requests participating in this circular dependency are in the idled state and at least one request's stimulation condition evaluates to true.

A subset of the model is enumerated and proven to be determinate. This is an important result because it proves that by limiting constructs and data dependencies determinate computation can be achieved with the model. Alternatively, by using the full expressive power of the model indeterminate computation may be modeled.

2.2 Problems to be Solved Using the Model

Solving problems with the model is the best way to gain experience and insights into its strengths and weaknesses. This section will begin by introducing the notation to be used in subsequent discussion and solutions.

2.2.1 Notation A BNF syntactic description is the appropriate vehicle for conveying the notation. Some liberties are taken in an effort to capture the essence of the model needed for problem solving. In the BNF to follow terminals are presented in uppercase and non-terminals in lower case. N represents the natural numbers, R the real numbers and λ the NULL production. A rigorous grammar is developed for the implementation project.

Data objects are defined by specifying their longevity, type, and designator as shown in figure 2-4. Other portions of the object definition will be contained within the text of the solutions' discussion. Structure types will be defined to be of *"stype"* which will be left to intuition in the examples. Designators are restricted to user name, spatial coordinates and relative and absolute sequence numbers. Initial values are informally specified to be allowed only for scalar objects of static longevity.

<dataobject> ::= <longevity> <type> <namelist> <initialvalue> ;
<longevity> ::= FIXED | STATIC | DYNAMIC | FLUID
<type> ::= INT | REAL | BOOLEAN | CHAR | STRUCTURE <stype>
<stype> ::= STRING
<namelist> ::= STRING <spatial> | <namelist>, STRING <spatial>
<spatial> ::= (<spatiallist>) | λ
<spatiallist> ::= <spatiallist> , N | N
<initialvalue> ::= = N | = R | = CHARACTER | λ

Figure 2-4. Data Object Definition Notation

A data object may subsequently be referenced by its designator:

<designator> ::= <username> <spatial> <instance>
<username> ::= STRING
<spatial> ::= (<spatiallist>) | λ
<spatiallist> ::= <spatiallist> , N | N
<instance> ::= ..<sequence> | λ
<sequence> ::= <relative> | <absolute>
<relative> ::= $+N$ | $-N$
<absolute> ::= N

Figure 2-5. Data Object Reference Notation

A request is given as:

<request> ::= <label> <s_e> <name> (<actuallist>) <t_e>;
<label> ::= STRING : | λ
<name> ::= <designator>
<actuallist> ::= <materials> ; <results>
<materials> ::= <matlist> | λ
<matlist> ::= <matlist>, <designator> | <designator>
<results> ::= <reslist> | λ
<reslist> ::= <reslist>, <designator> | <designator>
<s_e> ::= [<boolean expression>] | λ
<t_e> ::= [<boolean expression>] | λ

Figure 2-6. Request Notation

For some examples the detail of an action is superfluous, however the internal termination is interesting. To allow this type of abstraction the *actuallist* production will be expanded:

<actuallist>	::=	<materials> ; <results> <t_i>
<t_i>	::=	[<boolean expression>] | λ

Figure 2-7. Undetailed Request Notation

An action is detailed with the syntax:

<actiondetail>	::=	DETAIL <name> (<formallist>) <actionbody>
<formallist>	::=	<s_i> <materials> ; <results> <t_i>
<actionbody>	::=	{ <objectdef> <requestset> }
<objectdef>	::=	<objectlist> | λ
<objectlist>	::=	<objectlist> <dataobject> | <dataobject>
<requestset>	::=	<requestset> <request> | <request>
<s_i>	::=	[<boolean expression>] | λ
<t_i>	::=	[<boolean expression>] | λ

Figure 2-8. Detailed Action Notation

2.2.2 Arithmetic Problems The first set of problems presented involve simple arithmetic. The first problem is to compute the area of a circle. The model easily supports this computation:

```
static real Pi = 3.14;
static real r, Temp, Area;

R1: Mult(Temp, Pi; Area);
R2: Mult(r, r; Temp);
R3: Assign(7; r);
```

Figure 2-9. Area of a Circle Computation

This example requires sequential computation. The computation is initiated and the request labeled R3 is immediately eligible for execution. Upon termination the object *r* is available and R2 may be executed. Its completion satisfies all data dependencies on R1 and thus it is executed last. Notice that for this example, and the next, the requests are presented in the opposite order of their execution. This is done to emphasize the independence of presentation from execution.

Computing the length of the hypotenuse of a right triangle with the Phythagorean theorem exemplifies parallel computation. The solution is expressed as:

```
static real a, b, c, temp1, temp2, temp3

R1: Sqrt(temp3; c);
R2: Add(temp1, temp2; temp3);
R3: Mult(a, a; temp1);
R4: Mult(b, b; temp2);
R5: Assign(3; a);
R6: Assign(4; b);
```

Figure 2-10. Phythagorean Theorem Computation

This example allows parallel computation of the pairs <R5, R6> and <R3, R4>. Sequential execution of R2 and R1 follows. The management of the temporaries could be left to the model with the use of dynamic objects:

```
static  real a, b, c;
dynamic real temp;

R1: Sqrt(temp..3; c)
R2: Add(temp..-1, temp..+0; temp..+1)
R3: Mult(a, a; temp..+1);
R4: Mult(b, b; temp..+1);
R5: Assign(3; a);
R6: Assign(4; b);
```

Figure 2-11. Phythagorean Theorem with Dynamic Objects

Note that the introduction of the absolute incarnation on request R1 is not elegant. What is needed is the ability to stimulate R1 at the completion of R2, and consequently to use the relative temporary. Dataflow languages avoid this problem with the single assignment rule (which is simply a return to figure 2-10). Assuming that Add is a primitive action, the addition of a dummy synchronization object is not feasible. This problem is the topic of a related thesis [Yuk86] and is not examined further.

These solutions demonstrate that both sequential and parallel arithmetic computation can be performed within the model. The Phythagorean Theorem example demonstrates the bulk, and consequently error prone process of the introduction of temporaries when a computation must "rendezvous". A compiler should be capable of generating code to realize this tedium from a well formed expression. As such, the complexity of expression should not be attributed to the model rather it should dictate the need for clear syntax in a programming language based on the model.

2.2.3 Definite and Indefinite Looping This section examines some common problems which result in the need for definite or indefinite loops. The first problem examines the generation of a sequence of primes. The sequence ends when the first prime greater than 1000 is encountered. Assume an action *NextPrime* exists which given a natural number returns the next largest prime number.

```
dynamic int Prime;

R1: assign(1, Prime..1);
R2: NextPrime(Prime..+0; Prime..+1 [Prime..+0 < 1000]);
```

Figure 2-12. Finding a Sequence of Primes with Iteration

This example utilizes the iteration construct of the model to solve this problem. Upon termination of R2 the object Prime will contain the sequence. Referencing Prime..1 would obtain the first value, Prime..2 the second, etc. Looking in the other direction, Prime..+0 references the smallest prime greater than 1000, Prime..-1 the largest prime less than 1000, etc. Three problems suggest themselves from this example.

1. At what point does Prime..+1 become Prime..+0? Restated, when does the next prime to be created become the current prime? The example suggests that this identity change

occurs after binding of values to result list objects. This seems reasonable and will be informally accepted.

2. Are incarnations of Prime available prior to the completion of R2? This certainly seems beneficial to potential overlapped processing. [Fis87] investigates this issue. Another possible solution to this problem will be shown in chapter three.

3. Is there a method for examining the sequence? Assume that there is no way to tell the length of the generated sequence prior to running the iteration. One solution is to be able to access dynamic objects in the sequence they are generated in a time independent, but data driven manner. This will be revisited in chapter three. Another solution is to keep a count of the number of iterations and use this for subsequent iterations through the sequence. This solution suffers from the same problems which the next example will expose.

The next type of iteration to be examined regards recurrence relations. Consider the recurrence relation $I_n = I_{n-1} * n$. Generating this sequence until n exceeds 100 appears to be easily modeled by:

```
dynamic int I, n;

R1: Mult(I..+0, n..+0; I..+1 , n..+1 [n..+0 < 100]);
R2: Assign(1; I..1);
R3: Assign(2; n);
```

Figure 2-13. Modeling a Recurrence Relation with Iteration

However, the definition of iteration is too strict to permit this usage. A related thesis [Fis87] discusses the problems of iteration and proposes a broader definition. As such, this topic will not be further investigated.

2.2.4 Linear Algebra Problems Computer graphics often require operations on matricies to perform projection, rotation, and translation operations. The ability to rapidly perform these operations is imperative to real time graphic display such as animation. The model easily permits initialization of a matrix through the use of repetition. In the next example repetition expands Assign into one thousand separate, independent requests.

```
static int A(10,10,10), B(10,10,10)
Assign(A, B);
```

Figure 2-14. Matrix Initialization with Repetition

The model also easily supports matrix addition, also generating one thousand requests.

```
static int A(10,10,10), B(10,10,10)
Add(A; B)
```

Figure 2-15. Matrix Addition through Repetition

Matrix multiplication is presented in the same form as the above initialization and addition requests. However, the expansion to one thousand requests is entirely inadequate. [Fis87] further motivates this problem with repetition and proposes a solution.

2.2.5 Consumer and Producer Problems The consumer and producer problem represents a relationship that occurs often. Consider a process P which is producing *lines* and sending these to a lineprinter:

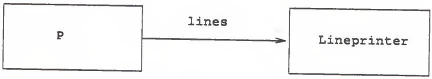


Figure 2-16. Process Sending lines to a Lineprinter

This problem exposes three troubles with the model:

1. There exists no control mechanism to allow an event driven process such as Lineprinter.

2. Assuming that multiple lines are produced it is not clear when these become known to Lineprinter. All lines might not be made available until P terminates or each line might be available as it is produced.

3. In the latter case of the last problem, if Lineprinter accesses the most recent version of lines and P produces these faster than Lineprinter can consume them, some will be lost. On the other hand, if Lineprinter consumes lines faster than P can produce them, the same line may be consumed multiple times.

Chapter three of this thesis will present two additions to the ACM model to address the producer and consumer problems. The first is a a new construct which allows a request to be executed multiple times. The second augments the dynamic longevity object to allow access to these types of objects in terms of a FIFO queue.

2.2.6 Readers and Writers The readers and writers problem is a useful characterization of concurrency problems which occur in database and operating systems. The problem to be solved is to allow multiple readers simultaneous access to an object, however writers require exclusive access. In addition, an object cannot be read before it is written. Extensions to this problem involve fair scheduling to prevent a writer from "starving" when there are frequent read requests.

The ACM model is able to handle the readers and writers problem very naturally. In the following figure the external stimulations (S1-S4) placed on the requests are added to denote the unspecified order in which these requests may be demanded.

```
dynamic int a;

R1: [S1] Reader1(a..+0 ;)
R2: [S2] Reader2(a..+0 ; )
R3: [S3] Writer1( ; a..+1)
R4: [S4] Writer2(  ; a..+1)
```

Figure 2-17. Readers and Writers Problem

This solution works well as long as the binding of a value to an instance of a dynamic object is an atomic action at the implementation level. This is not an unreasonable requirement to impose. Data drive will prevent reading of a data item which has yet to be written. The fairness problem is a question of the implementation's fairness to any request eligible for execution. This is a very pleasing solution to a difficult problem.

However, if the data type is extended to include spatial coordinates problems begin to arise. Assume that a reader wishes to read an entire vector and that multiple writers each fill an element of the vector (possibly through a repetition construct). The first attempt will be to declare the vector to be a dynamic longevity object.

```
dynamic int a(10);

R1:  Reader1(a..+0 ; );
R2:  Writer1(; a(1)..+1);
R3:  Writer2(; a(2)..+1);
      .
      .
R11: Writer10(; a(10)..+1);
```

Figure 2-18. Vector variant of the Readers and Writers Problem

The question at hand is to what is the sequence number bound? If it is bound to each individual element of the vector then Reader1 will not execute until requests R2-R11 have completed. If the sequence number is bound to the entire vector then R1 is eligible as soon as any one of R2-R11 completes. However, only the vector involved in the completed request will have a value.

The conclusion drawn from this example is to bind the sequence number to the whole object.

Consider augmenting the previous fragment as follows:

```
dynamic int a(10);
static  int sync;

R1:  Reader1(a..1 ; );
R2:  Writer1(; a(1)..+1);
R3:  Writer2(; a(2)..+1);
     .
     .
R11: Writer10(; a(10)..+1);

R12: Update(a(1)..+0; a(1)..+1, sync);
R13: Reader2(a(1)..+0, sync ; )
```

Figure 2-19. Update to a Dynamic Vector

Request R12 is dependent on the completion of R2-R11 at which time it is eligible to update the first element of the vector. The sync object is a dummy introduced to allow Reader2 to be started only after the Update request completes. The question becomes, what value of A(1)..+0 is obtained in request R13? Based on the strategy previously adopted it would be a(1)..1. However, since this scenario mimics a direct access file's population and subsequent update, the expected value would be A(1)..2 (the value updated in request R12).

The problems presented here with vectors and those presented under the Linear Algebra section are difficult. Additional research needs to be performed to determine clean solutions to the problems of objects with spatial coordinates.

2.2.7 Partialing a Computation The UNIX* Operating System Release V provides the make

* Registered trademark of AT&T

program to simplify the job of software construction and maintenance. The program accepts a specification file, known as a makefile, which elaborates the dependencies and instructions for constructing one or more products. Conventionally code which is useable by many different products is placed into a global library and that which is specific to a given product is kept locally. Header files contain data structure definitions, magic numbers, and other items which may have global interest.

When a header file changes the normal make sequence is to first remake all of the global libraries and then remake all products. Products may change because of library changes or the header file change, or both. If a product only depends on the library, it does not need to be compiled but only linkedited with the updated library.

In this problem it would be efficient to have products which are known to require compilation to run in parallel with the library's compilation. Once the library compilation completes linkediting of the library's users could begin. For this example assume that the materials to the requests are the changed objects and the results are the changed products.

dynamic structure file header, library, commands;

makelib(header; library);
makecmd(header, *library*; commands);

Figure 2-20. A Partialing Approach to Make

By partialing the library object in the makecmd request, the commands that require recompilation based on the header file change can run in parallel with the make of the library. Partialing represents a helpful mechanism for expressing explicit parallelism.

2.2.8 Detailing Example The server and client is a frequently encountered relationship. This relationship is normally characterized by a co-routine, a point in time when the client requests a service and waits for the response from the server. In this simplifying case assume that there is a single server and a single client.

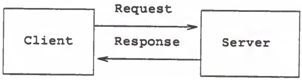


Figure 2-21. Client-Server Relationship

A first attempt at modeling this is:

dynamic int request, response, someinput, someoutput;

Client(someinput, response; someoutput, request);
Server(request; response);

Figure 2-22. First Attempt at Client-Server

This is incorrect since Client will require both materials prior to initiation. The response material will not be available until the request to the Server is made and the Server terminates. This is easily remedied by partialing response:

dynamic int request, response, someinput, someoutput;

Client(someinput, *response*; someoutput, request);
Server(request; response);

Figure 2-23. Second Attempt at Client-Server

However, the next problem is that generation and binding of request is tied to Client's termination. One way to avoid this problem is to extend the model to allow partialed results. This makes the model incredibly flexible (arguably too flexible). Two other possibilities exist which do not require extension of the model.

The first possibility is to split the Client into two components. The first component contains the requests up to the request for service. The second component runs after the Server's response is received.

```
dynamic int request, response, someinput, someoutput;

ClientPart1(someinput; request);
ClientPart2(response; someoutput);
Server(request; response);
```

Figure 2-24. Two Part Client and Server Solution

This solution is satisfactory so long as extensive information is not necessary to be passed between the two portions of the client and no activity may occur in the client while being serviced. An almost identical solution is to detail the problem and include the server as a request within the detailed action's request set. This solution avoids the problems with the two part client solution.

```
DETAIL    clientserver(someinput; someoutput);
{
          /* some requests */

          server(request; response);

          /* some other requests */
}
```

Figure 2-25. Detailing Solution to Client-Server

It's hard to be dogmatic about which of the previous two approaches is best in the general case. Since the intent of this portion of the study has been to gain experience with the model, it is reassuring to know that not only can this problem be solved within the model but that it can be solved using different methods.

2.2.9 Transaction Processing The Client-Server model examined in the last section is an excellent model for a transaction processing system. For this problem the client will be an accountant's entry system, the request will be a debit or credit transaction to a general ledger database. The server will be the database manager for the ledger and the response will indicate the transaction's success. The fictitious company has business sense enough to have only one general ledger and enough programming experience to fear concurrent updates.

To complicate this example, assume that there are multiple clients (accountants) interacting with the server (database manager). Each is an identical copy and the detailing approach described previously has been taken. (The two part client approach will also work but requires the unveiling of the concepts of chapter three.) The company's accounting department has three accountants and is modeled by the request set:

```
dynamic structure transaction ledgeritem;
dynamic structure database generalledger;

Accountant(ledgeritem, generalledger; generalledger);
Accountant(ledgeritem, generalledger; generalledger);
Accountant(ledgeritem, generalledger; generalledger);
```

Figure 2-26. Request Set for Accountanting

The detail for an Accountant action is:

```
DETAIL          Accountant(item, db; db)
{
                fluid boolean doit, ack;

                Validate(item; doit);

[doit = true]   Updatedb(item , db; ack, db);

[ack = false]   Fail();
[doit = false]  Fail();  /* Fail() is described in [Yuk86] */

}
```

Figure 2-27. Detail of Accountant Action

Updatedb is the request for the server action. Because there are multiple requests for Accountant in the original request set it is possible to have multiple Updatedb servers running simultaneously. The readers and writers problem examined previously showed that the model would guarantee that the results would be consistent (ie two writers could not be writing on the same dynamic longevity object simultaneously). However, databases also need to be conservative in that all transactions applied must be reflected in the most recent version.

The underlying problem is the serialization of the transactions on the database. The replication component of the corporality of an object will be utilized here. There are two simple methods to do this, both of which fall within the model. The first is to restrict the Updatedb action to only one replication. This will insure that only one update is running on the database at a time. However, if Updatedb is part of a generalized database package this solution may have the undesirable side effect of allowing only one of the company's databases to be updated at a time.

The second solution is to restrict the generalledger database to only one replication. In this manner only one Updatedb request can satisfy the data drive principle and perform a transaction at a time. This is a very attractive solution from both a modeling and practical standpoint.

2.2.10 Dining Philosophers The dining philosophers problem is a classic resource allocation problem. The problem forces concurrent processes to cooperate in the use of shared resources. Lack of cooperation can lead to deadlock. The problem is best described pictorially:

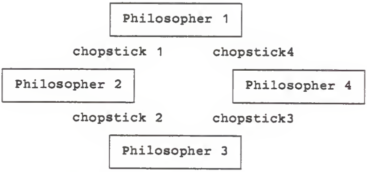


Figure 2-28. Dining Philosopher Problem

A philosopher can either think or eat. Once the decision to eat has been made both chopsticks must be acquired. The replication scheme will provide a simple mechanism for solving this problem. A restriction of one replication will be placed on each chopstick. Assume that a mechanism has been developed (see Chapter 3) which allows a request to return to the idle state after it has normally terminated.

```
fluid boolean Hungry1, Hungry2, Hungry3, Hungry4 = false;
static boolean Chopstick1, Chopstick2, Chopstick3, Chopstick4 = true;

/* Thinktime randomly sets Hungryx to true */
[Hungry1 = false] Think( ; Hungry1);
[Hungry2 = false] Think( ; Hungry2);
[Hungry3 = false] Think( ; Hungry3);
[Hungry4 = false] Think( ; Hungry4);

/* Eat will randomly set Hungryx to false */
[Hungry1 = "true"]  Eat(Chopstick1, Chopstick2; Hungry1);
[Hungry2 = "true"]  Eat(Chopstick2, Chopstick3; Hungry2);
[Hungry3 = "true"]  Eat(Chopstick3, Chopstick4; Hungry3);
[Hungry4 = "true"]  Eat(Chopstick4, Chopstick1; Hungry4);
```

Figure 2-29. Dining Philosophers Solution

This is a very clean solution to this problem. The Hungry flip-flop is used to hold the Philosopher's state. When set, the availability of the single replication of each chopstick (the shared resources) drives the execution of the Eat request. The model will inherently prevent deadlock.

2.2.11 Physical Locality Returning to the accounting example there is the need in a physical system to map each Accountant request to a physical (or logical) location. The model allows this through the location component of an object. This mapping isn't necessary for solving problems with the model, per se., but is crucial in a functional implementation.

A key insight gained in this research has been that given the capability to bind a modeled object to a physical object in the underlying implementation significant problems can be solved with the model in an abstract manner. For example, the dining philosopher's problem could map the Chopstick objects into physical resources. In the transaction processing model the database need only be modeled by an atomic type with a mapping to an actual database file provided. This insight is important because it significantly reinforces the flexibility of the model, allows complexity of a given implementation's environment to be abstracted, increases portability of the

models (the concept of a serializable database update approach is the same whether the database is IMS or a raw disk partition), and allows introduction into existing environments.

2.2.12 Summary This section has posed some modeling problems for solution with the ACM model. The solutions obtained were instructive in that a few deficient areas were encountered, significant experience and insight into the model's expressive power was gained and reported, and finally some very difficult problems were found to be easily solved. The expressive power of the model is generally pleasing, and will increase with the augmentations suggested in the next chapter and in [Fis87] and [Yuk86]. The idea of presenting numerous problems and solutions is due to Hoare's treatment of CSP in [Hoa78].

CHAPTER 3

3. Extensions to the ACM Model

The last chapter's presentation raised a few areas for further examination. This chapter presents a new construct and a new grouping of objects as answers to a few of these problem areas.

The first section presents a new construct to support looping. This construct allows indefinite looping with rebinding of materials and results on each invocation of the loop. The second section allows for pipelined access to objects of dynamic longevity. Combining the two proposals allows the modeling of very complex systems. The final section will present a banking model. This model contains a multiserver queue, serialization of a series of database transactions, and the subsequent routing of acknowledgements to be merged with the original transaction.

3.1 Loop Construct

3.1.1 Motivation Chapter two discussed the producer and consumer problem in terms of a process sending lines to a lineprinter. The problem is stated here in its generalized form.

> A *Producer* process produces *goods* which are sent to a *Consumer* process. The relative speed of the Producer and Consumer is unknown as is the number of goods to be produced.

The first step is to examine existent constructs to determine if any of them can be used to implement the Consumer. A simple request clearly cannot be used since the total number of

goods would need to be known and each would need to be reflected in the Consumer's material list. Repetition fails for the same basic reason, the number of goods is unknown. An additional problem with repetition occurs when the relative order of incoming goods is important (as it would be for a lineprinter). The nature of repetition is to execute numerous independent requests simultaneously. Since the relative running times of each of these requests cannot be ascertained, the order may not be conserved.

Iteration, at first glance, appears to be the needed construct. Iteration would allow repeated, sequential execution of the Consumer. However, the definition of iteration stipulates a relationship of the materials for an iteration cycle on the results of a previous iteration cycle [Ung78a]. A startup period is allowed in which this relationship doesn't hold and materials may be gathered from the external environment [Fis87]. A potential solution is to expand the startup period's length to be proportional to the number of goods to be consumed. This is inelegant because it inhibits parallelism by requiring the Producer to run to completion prior to the Consumer's initiation.

To remedy this problem a loop request is needed. The requirements for this construct are:

1. Upon successful completion the request must be eligible for subsequent re-execution. Termination of the request should only occur upon the request's failure or upon success of the external termination condition.

2. Materials must be rebound on each invocation of the loop. This allows objects which change external to the loop to be reflected in each invocation of the loop.

The next section presents four definitions which satisfy these requirements.

3.1.2 Definitions

Definition 3.1-1 A loop construct is a request R which may be transitioned from the idle state to the enabled state, zero or more times. The loop construct is referred to as a *loop request*. A loop request will be denoted by preceding the request name with an asterisk.

Definition 3.1-2 A loop request R is transitioned from the idle state to the enabled state in the same manner as a non-loop request. In other words, data drive and the truth of the external stimulation govern this state transition.

Definition 3.1-3 The state for a loop request is determined upon the request's termination as:

idle	if t_e evaluates to false, or
disabled	if t_e evaluates to true, or
disabled	if the request fails (see [Yuk86]).

Definition 3.1-4 For a loop request which is to be transitioned from the idle to the enabled state the designators of all materials are evaluated. Objects with longevity dynamic or fluid may assume different values on each invocation of the loop.

3.1.3 Discussion The definitions of the previous section taken together satisfy the requirements previously stated. Definition 3.1-1 poses the only real difference between a loop request and other requests. This difference, the capability for multiple executions, is governed by the state transition criteria specified in definition 3.1-3. Definitions 3.1-2 and 3.1-4 are no different than those for non-looping requests, and are presented for completeness.

Returning to the producer-consumer problem consider the following code fragment:

```
fluid int goods;

        *Producer(; goods);
        *Consumer(goods; );
```

Figure 3-1. Producer-Consumer Modeled with Loop Requests

In this fragment, Producer is destined to run forever producing goods. Consumer is also non-terminating, but will wait until at least one good has been produced. A serious drawback to this example as presented is that the relative speed of Producer and Consumer will dictate whether the Consumer misses some goods, re-consumes the same goods, consumes the "correct" goods, or a combination of the three. This synchronization problem will be addressed in the next section. For the remainder of this section correct synchronization will be assumed.

If it is desirable to terminate, an external termination can be added to both requests. Assume that when Producer internally decides to terminate the resultant value of goods will be zero.

```
fluid int goods;

        *Producer(; goods) [ goods = 0];
        *Consumer(goods ;) [ goods = 0];
```

Figure 3-2. Producer-Consumer with Termination

Another possibility is that there are two consumers, one which consumes only goods of value one and the other consumes all positive valued goods. Goods which have negative value are not consumed.

```
fluid int       goods;
                *Producer(; goods) [ goods = 0];
[goods = 1]     *Consumer(goods; ) [ goods = 0];
[goods >= 1]    *Consumer(goods; ) [ goods = 0];
```

Figure 3-3. Producer-Consumer with Filter

This fragment exemplifies a filtering effect which is achieved through the external stimulation. This filtering effect can be used to achieve event driven processing. The above example will cause both consumers to execute when the value of goods is one, one consumer to execute when the value is greater than one, and none will execute when the value is negative. All three requests terminate when the value of goods is zero.

The looping request can be used to implement a conventional loop. However, care must be exercised in specifying the external termination condition. By definition, the truth of the external condition of a request will cause the request's state to be changed to disabled. If the termination condition is based on an object other than one contained in the request's result list, processing may be terminated while the request is enabled.

In the examples above it might be necessary to perform some wrapup processing (such as producing a burst page from the lineprinter) prior to termination of the consumer request. The consumer action is augmented to propagate its received goods upon completion. Figure 3-2 is changed to reflect this as the basis for the termination condition.

```
fluid int   goods,processed1;

     *Producer(; goods) [ goods = 0];
     *Consumer(goods; processed1) [ processed1 = 0];
```

Figure 3-4. Producer-Consumer Avoiding Premature Termination

3.2 Pipelined Dynamic Objects

3.2.1 Motivation The previous section's introduction of the loop request raised the issue of synchronization in the producer and consumer example. The primary problem is that the speed with which the producer produces and the consumer consumes cannot be unilaterally ascertained. If the producer produces goods faster than the consumer can consume them, they are lost.

Alternatively if the consumer consumes goods faster than they are produced, the same goods may be re-consumed. Often times both phenomena occur and some goods are lost and some consumed multiple times.

3.2.2 Solutions The first solution to this problem lies within the model (with the loop request augmentation of the previous section). This solution introduces a synchronization object which flip-flops between the producer and consumer. In the following example the object turn is set to P when the Producer should produce and to C when a good has been produced and the Consumer should consume. An initial seeding is needed to get the system started.

```
fluid int       goods
fluid char      Turn

                assign('P', Turn);

[Turn = 'P']    *Producer(; goods, Turn);
[Turn = 'C']    *Consumer(goods; Turn);
```

Figure 3-5. Synchronization with a Flip-Flop

This solution works very well but at the expense of parallelism. It would be beneficial to allow Producer to produce the next good while consumer was consuming the last good. To attain this overlapped processing the next solution adds memory to the Flip-Flop synchronization. A new Propagate action is introduced which copies the incoming object, resets a flip-flop indicating that its predecessor may produce, and sets a flip-flop indicating that its successor may consume. The detail for Propagate is given in figure 3-6. Figure 3-7 pictorially represents the producer and consumer problem with the Propagate request. Figure 3-8 models the system.

```
DETAIL Propagate(objectin; objectout, Pturn, Cturn)
{
     Assign('P'; Pturn);
     Assign('C'; Cturn);
     Assign(objectin; objectout);
}
```

Figure 3-6. Propagate Action Detail

Figure 3-7. Synchronization Through Explicit Buffering

```
fluid int     goods1,goods2;
fluid char    Pturn, Cturn;

              assign('P'; Pturn);
              assign('P'; Cturn);

              [Pturn = 'P'] *Producer(; goods1, Pturn);

              [Pturn = 'C' AND Cturn = 'P']
                    *Propagate(goods1; goods2, Pturn, Cturn);

              [Cturn = 'C'] *Consumer(goods2; Cturn);
```

Figure 3-8. Synchronization with Propagate Action

An analysis is in order to demonstrate the parallelism added to the system. A table will be used to depict the value of objects and the state of actions over time. Time periods for which state or value changes do not occur will not be enumerated.

First, consider the flip-flop solution of figure 3-5. Assume that three time periods will be required to produce an object, four time periods to consume it, and one to perform state transitions. Figure 3-9 performs the analysis of the flip-flop solution. Twenty three time periods are required before the Consumer begins to consume the third good.

Figure 3-10 performs the same analysis for the propagation solution of figure 3-8. In this analysis the Propagate action is assumed to require only one time period. This analysis shows that only twenty one time periods are required before the Consumer begins to consume the third good. Overlapped processing of the Producer and Consumer occurs at time periods 7 through 9, 14 through 16, and 21 in the analysis.

Time	Turn	goods	Producer	Consumer
0	P	-	Idle	Idle
1	P	-	Enabled	Idle
4	C	1	Idle	Idle
5	C	1	Idle	Enabled
9	P	1	Idle	Idle
10	P	1	Enabled	Idle
13	C	2	Idle	Idle
14	C	2	Idle	Enabled
18	P	2	Idle	Idle
19	P	2	Enabled	Idle
22	C	3	Idle	Idle
23	C	3	Idle	Enabled

Figure 3-9. Analysis of Figure 3-5

Time	Pturn	Cturn	goods1	goods2	Producer	Propagate	Consumer
0	P	P	-	-	Idle	Idle	Idle
1	P	P	-	-	Enabled	Idle	Idle
4	C	P	1	-	Idle	Idle	Idle
5	C	P	1	-	Idle	Enabled	Idle
6	P	C	1	1	Idle	Idle	Idle
7	P	C	1	1	Enabled	Idle	Enabled
10	C	C	2	1	Idle	Idle	Enabled
11	C	P	2	1	Idle	Idle	Idle
12	C	P	2	1	Idle	Enabled	Idle
13	P	C	2	2	Idle	Idle	Idle
14	P	C	2	2	Enabled	Idle	Enabled
17	C	C	3	2	Idle	Idle	Enabled
18	C	P	3	2	Idle	Idle	Idle
19	C	P	3	2	Idle	Enabled	Idle
20	P	C	3	3	Idle	Idle	Idle
21	P	C	3	3	Enabled	Idle	Enabled

Figure 3-10. Analysis of Figure 3-8

The construct built in figure 3-8 is a pipeline. Attributes of the pipeline are:

- *N* objects may be in the pipeline simultaneously. *N* is the number of Propagate actions.
- The reader and the writer of the pipeline will, at times, overlap in their processing.
- The reader of the pipeline will receive objects in the exact sequence the writer produced them.
- The reader will never obtain the same object twice (unless the object is written twice by the writer).
- The reader will never miss an object.

The pipeline solution, combined with the loop request presented in the last section of this chapter, allow robust solutions to the Producer-Consumer and Prime number sequence problems presented in the second chapter. By varying the number of Propagate actions the memory

requirements and parallelism may be controlled.

Now consider the Client-Server presented in the second chapter. One of the situations analyzed was the transaction processing environment. There, multiple clients (Accountants) were issuing transactions to a database management server. Transactions were serialized by restricting the general ledger database to a single replication. Modeling-wise this solution appeared fine, however the sequencing of transactions was never (and could not be) explicitly stated. To keep the shareholders and auditors happy, the fictitious company wishes to insure that transactions are applied to the database in the order that they are received.

The pipeline described above seems to solve this problem, assuming some method can be determined to merge the transactions of multiple clients into a single pipeline. This task is non-trivial and might be achieved through the use of dynamic objects and synchronized actions. If a solution exists, it will undoubtedly be tied to the number of clients. Adding a new client will require rework of the synchronization objects.

In addition to the multiple client case, it would be desirable to model the multiple server case. To be totally flexible, it would be beneficial to model the multiple clients supported by multiple servers. The pipeline construct appears to be the answer, the problem at hand is the fan in (multiple writers to the pipeline) and fan out (multiple readers). Again, if a solution exists the complexity will be unwieldy and very sensitive to the number of servers and clients. What is needed is a mechanism, within the model, to manage multiple reader/multiple writer pipelines.

Recalling the readers and writers solution in the previous chapter, it was pointed out that the model guaranteed conflict free creation of the different instances of dynamic longevity objects. Dynamic longevity objects are a convenient method for managing the multiple writer problem. By definition dynamic longevity objects will retain the values of their previous incarnations and

the incarnations will be ordered chronologically. This obviates the need for memory and the Propagate action as presented in figure 3-8.

The next section will extend the definitions relating to objects of dynamic longevity to accommodate the remaining problem, the multiple readers on the pipeline. The definitions specify a disciplined method within the model for inspecting and obtaining each incarnation.

3.2.3 Definitions

> *Definition 3.2-1* When a new incarnation of an object with longevity dynamic is created, the value of that incarnation is enqueued at the end of a FIFO list associated with the object.

> *Definition 3.2-2* When a material within a request references an object of longevity dynamic with *d..next*, where *d* is the designator of the object and *next* is a keyword, the value contained in the head element of the FIFO associated with the object is dequeued. If the FIFO is empty, this material is considered to be nonexistent.

> *Definition 3.2-3* When a reference to an object of longevity dynamic is made as *d..next* within a stimulation or termination condition, the head element of the FIFO is examined. The element is not dequeued. If the FIFO is empty, this object is considered to be nonexistent.

> *Definition 3.2-4* When a reference to an object of longevity dynamic is made as *d..empty* the value obtained is a boolean. The boolean is *true* if the FIFO associated with the specified object is empty. The boolean is *false* if the FIFO associated with the object is not empty.

3.2.4 Discussion The definitions of the previous section taken together with the definition of a dynamic longevity object cover the pipelined dynamic object. Definition 3.2-1 provides for the

enqueuing of each new incarnation as it is created. Definitions 3.2-2 specifies that referencing *d..next* within a material list of a request allows the next incarnation in the pipeline to be retrieved. This definition allows multiple readers to retrieve distinct incarnations from the pipeline without the need for explicit concurrency control.

The last two definitions allow non-destructive inspection of the pipeline. Definition 3.2-3 provides stimulation and termination conditions the capability to inspect the head element. This capability can be used, for example, to guard requests so that only a selected range of values will cause stimulation. Definition 3.2-4 defines the *d..empty* notation which may be used to determine if the pipeline is empty.

Chapter two discussed the prime number sequencing problem. One possible solution to accessing the entire generated sequence would be a request which begins upon successful completion of the NextPrime request and obtains each prime from the object pipeline until it is empty. This would be realized as:

```
dynamic   int Primelist;

          Assign(1; Primelist);
NP:       NextPrime(Primelist; Primelist..+1 [ Primelist < 1000]);

[S(NP)]   *ListPrimes(Primelist..next; ) [Primelist..empty && S(NP)]
```

Figure 3-11. Solution to Prime Number Sequence

The next section will model a bank. This modeling will show the power available with the loop request and pipelined dynamic objects when modeling a non-trivial system.

3.3 Examples Using the New Constructs

The first portion of the bank to be examined is the teller and customer relationship. Customers arrive at an unspecified rate. This is modeled through a pure producer process. A fixed number

of tellers (four for this example) wait on customers. This configuration is referred to as a multiserver queue in queueing theory. This is pictured in figure 3-12.

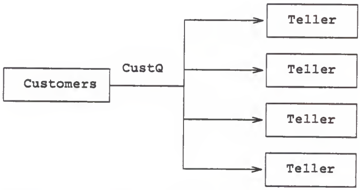


Figure 3-12. Customers and Tellers - A Multiserver Queue

The multiserver queue can be modeled as a pipeline with a single writer and multiple readers. Customers and Tellers are never ending processes. The Customers request will produce an object structured as a Ctransaction. This structure will encompass the deposit, withdrawal, loan payment, and other forms which are presented to the teller. This structure will include a unique customer id.

A solution for this configuration exists within the model with the augmentations provided in this chapter. Figure 3-13 gives this solution. Notice that structures from the CustQ are dequeued by Tellers as they are available. No explicit synchronization is necessary.

```
dynamic structure Ctransaction CustQ;

*Customers(; CustQ);
*Teller(CustQ..next; )
*Teller(CustQ..next; )
*Teller(CustQ..next; )
*Teller(CustQ..next; )
```

Figure 3-13. Customers and Tellers Model

Now, the teller must process the customer's transaction slip and turn this into a database transaction. These transactions are then presented to a database manager for processing. Use of a pipeline provides a convenient mechanism for serializing the transactions. This pipeline will be multiple writer and single reader. The teller and database interactions are shown in figure 3-14. The model is given in figure 3-15.

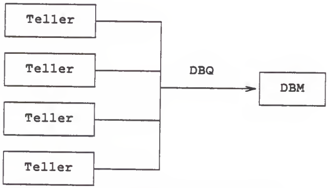


Figure 3-14. Teller-Database Manager Queue

dynamic structure Dbtransaction DBQ;

*Teller(; DBQ);
*Teller(; DBQ);
*Teller(; DBQ);
*Teller(; DBQ);
*DBM(DBQ..next;);

Figure 3-15. Teller-Database Manager Model

Since it is desirable to merge the database acknowledgments with the customer's original transaction a merge step is necessary. To model this each teller will run as two steps. The first step consists of receiving the customer's transaction slip and turning this into a database transaction (figure 3-15). This step will then pass the original customer transaction to the second step which will merge this with the database acknowledgement for this transaction. Figure 3-16 depicts the second step of this process. It is modeled in figure 3-17.

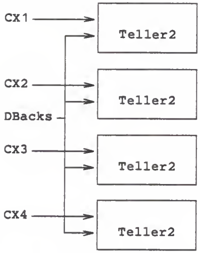


Figure 3-16. Merging Customer Transaction with Acknowledgement

```
dynamic struct Ack DBacks; /* From DBM */
fluid  struct Ctransaction CX1; /* From teller 1 */
fluid  struct Ctransaction CX2; /* From teller 2 */
fluid  struct Ctransaction CX3; /* From teller 3 */
fluid  struct Ctransaction CX4; /* From teller 4 */

[CX1.id = DBacks.id..next] *Teller2(CX1,DBacks..next; );

[CX2.id = DBacks.id..next] *Teller2(CX2,DBacks..next; );

[CX3.id = DBacks.id..next] *Teller2(CX3,DBacks..next; );

[CX4.id = DBacks.id..next] *Teller2(CX4,DBacks..next; );
```

Figure 3-17. Model for Transaction and Acknowledgement Merging

The customer transactions from the first portion of each teller are funneled to the second part through the fluid object CXn. When the customer id within the customer transaction matches the next element in the database manager's acknowledgement queue, the Teller2 request consumes these. Teller2 is a pure consumer representing the final portion of processing for the customer transaction.

All of the pieces developed above may be put together to form the final banking model of figure 3-18.

```
dynamic structure Ctransaction CustQ;  /* Customer's original transaction  */
dynamic structure Dbtransaction DBQ;   /* Database transaction queue       */
dynamic struct Ack DBacks;             /* Database manager ack. queue      */
fluid   struct Ctransaction CX1;       /* Customer xaction - first teller  */
fluid   struct Ctransaction CX2;       /* Customer xaction - second teller */
fluid   struct Ctransaction CX3;       /* Customer xaction - third teller  */
fluid   struct Ctransaction CX4;       /* Customer xaction - fourth teller */

*Customers(; CustQ);    /* Pure producer of customer transactions */

*Teller(CustQ..next; CX1, DBQ); /* First teller    */
*Teller(CustQ..next; CX2, DBQ); /* Second teller   */
*Teller(CustQ..next; CX3, DBQ); /* Third teller    */
*Teller(CustQ..next; CX4, DBQ); /* Fourth teller   */

*DBM(DBQ..next; DBacks);        /* Database manager    */

/* Pure consumers to merge customer transaction and acknowledgement */

[CX1.id = DBacks.id..next] *Teller2(CX1,DBacks..next; ); /* First teller  */
[CX2.id = DBacks.id..next] *Teller2(CX2,DBacks..next; ); /* Second teller */
[CX3.id = DBacks.id..next] *Teller2(CX3,DBacks..next; ); /* Third teller  */
[CX4.id = DBacks.id..next] *Teller2(CX4,DBacks..next; ); /* Fourth teller */
```

Figure 3-18. Banking Model

3.4 Summary

Two extensions to the model have been proposed and defined. The loop request allows a request to be performed multiple times. The pipelined dynamic object utilizes the buffered nature of dynamic objects as originally defined and augments this with a disciplined access method. Providing synchronization within the model, through the pipelined object, relieves a difficult burden from the programmer. The two, working together, allow modeling of difficult, non-deterministic problems as shown with the banking example.

CHAPTER 4

4. Implementation of a Subset of the ACM Model

This chapter will detail the design of a compiler for a language (SMART) which implements a portion of the ACM model. The first section defines the environment that the compiler is to run in and the environment it is to produce output for. The second section presents the language features implemented and the syntax for these features. The final section specifies the design for the compiler. This design includes the characteristics and execution model for the virtual machine for which the compiler produces output.

4.1 Implementation Environment

4.1.1 Development Environment The compiler for SMART and the operating system to support it will be developed on an AT&T 3B5 computer running UNIX System V. The language development tools, yacc and lex, will be used to facilitate the compiler's development. All of the development work will be done on Kansas State University computing facilities.

4.1.2 Operational Environment The operating environment for the final product is a network of five AT&T 3B2/300 computers each running the System V operating system. These machines are networked together through 3BNET, an Ethernet* based network. The AT&T 3B5 and AT&T 3B2 computers are object code compatible, so the products may be developed on the larger machine and then tested on the smaller ones.

The operating system built to support SMART will sit on top of the System V operating system.

---

* Registered trademark of Xerox Corporation.

Each 3B2 will be equipped with several identical processes:

- Network handler - Performs communication with the other machines over the network.
- Scheduler - Handles the initiation and termination of requests.
- Terminal handler - Two processes will supervise the user interface. One will handle input from the terminal and the other will write to the screen and log the session.
- Environment handler - This process will execute the compiler's output graph. One of these processes is started for each program activated.

Figure 4-1 gives a block diagram of the process architecture for a single machine. These processes will communicate through the message interprocess communication facility provided by the System V operating system. As mentioned above, this same process structure will be replicated on each machine. This allows a homogeneous view of the system. (In other words, there will not be a single master machine.)

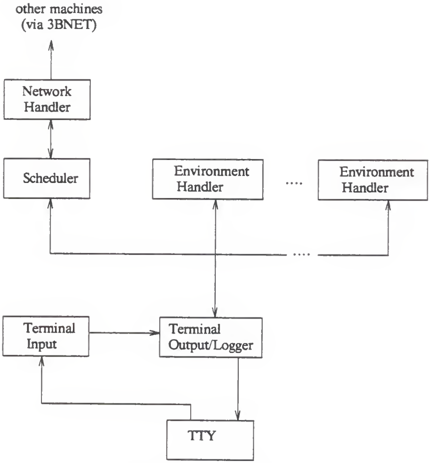


Figure 4-1. Architecture for SMART Implementation

4.1.3 Virtual Machine Environment The compiler for SMART will build a graph to be executed by the environment handler of figure 4-1. A very important technique in building the compiler is to shield it from knowledge of the underlying implementation of the system it is to run on. What will be done with SMART is to compile the language into a virtual machine format. This is similar to Pascal's P-Code approach to the execution environment. Figure 4-2 shows the path from the original SMART code to its execution. The intent of this pathway is to build the compiler, a relatively large amount of code, in a machine independent form. The machine

peculiarities (architecture, instruction set etc.) can then be exploited in a virtual to real machine transformational program. The real machine format is finally read and executed by the environment handler.

Section three of this chapter presents a more detailed view of the virtual machine's characteristics and execution strategy.

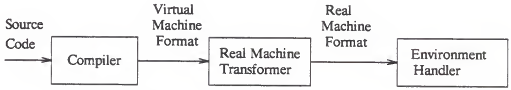


Figure 4-2. Language to Execution Transformations

4.2 The SMART Language

4.2.1 Relationship to the ACM Model The ACM model presented and augmented in the previous chapters is the basis for the SMART language. The following components of the model were included in this language design:

- Data objects of longevity static, dynamic, and fluid. These objects may be of the simple types real, integer, or character.
- Requests at the "atomic" level. In other words, detailing was not implemented.
- External stimulation, termination and internal termination conditions. These are constructed with arbitrarily complex boolean expressions.
- Success/failure conditions ([Yuk86]).
- Iteration ([Fis87]).

- Indefinite looping (chapter 3).

- Pipelined dynamic objects (chapter 3).

4.2.2 Syntax for SMART The syntax for SMART is very similar to that presented in chapter two. The construction of the syntax was greatly facilitated by the yacc parser generator tool. Utilizing a parser generator not only simplifies the task of developing the compiler, but also provides a rigorous check of the grammar for ambiguities. The grammar which is used by yacc to build the parser is contained in Appendix I.

4.2.3 Lexicon for SMART The lexical analyzer will be generated by the lex tool. The lexical analyzer will treat the following "words", regardless of case, as reserved:

CHAR	DYNAMIC	F	FILE
FIXED	FLUID	INT	REAL
S	STATIC	VAR	

Figure 4-3. SMART's Reserved Words

Punctuation, operators and other special characters are:

/	*	+	-	:	;	.
=	==	>=	>	<	<=	!=
[	]	(	)	&	&&	|
||	,					

Figure 4-4. Punctuation, Operators and Other Special Characters

Comments will obey the C language convention of /* */. These will be stripped by the lexical analyzer.

4.2.4 Static Semantics [Fis87] provides an informal description of the language's static semantics. The compiler implementation section lists the static semantic checks implemented.

4.3 Compiler Design

4.3.1 Overview The compiler will perform one pass over the input program. In this pass checks for lexical, syntactic, and, to some extent, semantic conformance of the input source to the language definition will be made. This pass will also build three data structures: the string table, symbol table, and condition lists. Figure 4-5 presents a block diagram of the front end of the compiler.

If no errors are detected after making a complete pass through the user's code, the back end of the compiler is invoked. The back end performs a few manipulations on the symbol table to build the virtual machine representation of the program. This is then written to the output file. Figure 4-6 is a pictorial representation of the backend of the compiler.

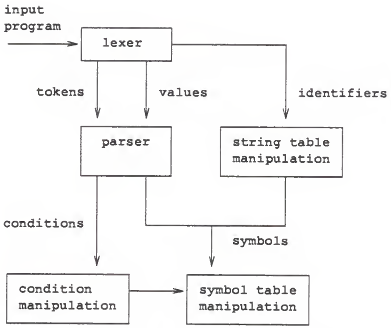


Figure 4-5. Block Diagram of the Compiler's Front End

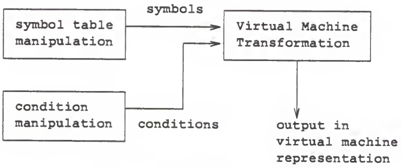


Figure 4-6. Block Diagram of the Compiler's Back End

4.3.2 Lexical Analyzer The lexical analyzer will be constructed with the *lex* lexical analyzer generator. The lexicon of the language is described with regular expressions which are mapped into token values. The lexicon for SMART was presented in the previous section. The generated lexical analyzer is an integer valued function which scans the input stream and returns a token when a member of the lexicon is recognized.

The lexical analyzer interfaces with the parser, the string table manipulation, and condition manipulation components. The interface with the parser consists of the recognized token (the integer value returned from the lexer) and a stack containing the token value. The tokens are specified within the yacc specification and are placed in the generated header file *y.tab.h*. The token values are placed into the union *YYLVAL*. This union is also specified in the yacc specification. Figure 4-7 specifies the members of this union to be populated by the lexical analyzer.

strptr	Pointer to a string table entry returned for an identifier (ID) token.
realval	Floating point value associated with the REALVAL token.
intval	Integer value associated with the INTVAL token.
charval	Character value associated with the CHARVAL token.

Figure 4-7. YYLVAL Union (Lexical Analyzer)

When an identifier is recognized it is placed into the string table. The string table is constructed as a hash table with doubly linked overflow lists. The current implementation only requires a singly linked list; a doubly linked list was built to facilitate future addition of detailing. Figure 4-8 depicts the string table layout and the structure of a string table element.

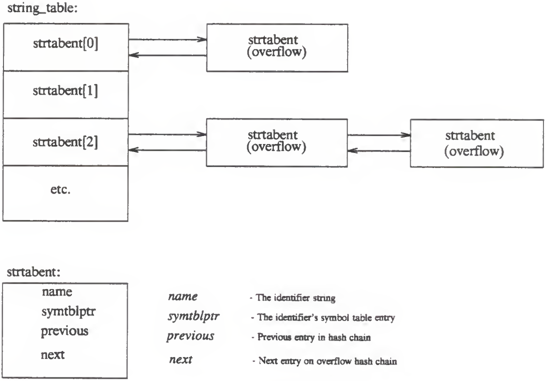


Figure 4-8. String Table Layout

4.3.3 Parser The parser for SMART will be built with the *yacc* parser generator program. Yacc constructs a parser, which implicitly interfaces with the lex generated lexical analyzer, based on a four part specification.

1. YYLVAL Union specification. This union is the value associated with terminals and non-terminals. The values of the terminals were previously defined in figure 4-7. Non-terminals can assume values as well; this provides a very convenient mechanism for constructing a tree structure in a bottom up, syntax directed fashion. Figure 4-9 describes the non-terminal values used in the YYLVAL union. The condition, sequence and dynamic structures will be described in the symbol table section.

2. Token specification. The tokens generated by the lexical analyzer must be specified. These and the YYLVAL union are placed into a generated header file, *y.tab.h*, which is compiled into the lexical analyzer to insure consistency. Tokens may be augmented with YYLVAL values in this portion of the specification. These tokens were enumerated in figure 4-7.

3. Nonterminal type specification. Nonterminals which have a YYLVAL value associated with them must be specified. Figure 4-10 lists the nonterminals and the YYLVAL union elements they are associated with.

4. Augmented Grammar specification. The heart of the parser specification is an augmented BNF grammar. The SMART grammar is contained in Appendix I. The augmentations are C language code which is executed upon a production's recognition. The SMART language parser will build the symbol table and condition list, in addition to performing semantic checks within the C language augmentations.

condptr	Pointer to a list containing (stimulation or termination) conditions. The list is stored in reverse polish notation.
seqptr	Pointer to a structure containing information on a dynamic longevity object's sequence component.
dynptr	Pointer to a list of symbol table pointers.

Figure 4-9. YYLVAL Union (Parser)

Nonterminal	YYLVAL element
condition	condptr
desig	strptr
instance	seqptr
longevity	intval
mlist	dynptr
relop	intval
rlist	dynptr
request	strptr
rspec	strptr
stype	intval
se	condptr
sequence	seqptr
sign	intval
te	condptr
ti	condptr
username	strptr

Figure 4-10. Nonterminals and Their YYLVAL Values

4.3.4 Symbol Table The symbol table is the primary data structure to be built by the compiler. This data structure is used by the front end to perform semantic checks. It is then massaged by the backend to produce the virtual machine representation. There are three types of symbol table entries:

1. Request entry,

2. Data Object entry, and

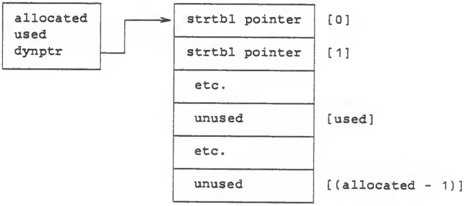
3. Label entry.

The request entry roots a tree of entries associated with a request. There are six pertinent pieces of information associated with a request entry. These are described in figure 4-11. The dynlist is simply a convenient mechanism for managing a variable sized array of string table pointers. The dynlist structure is depicted and described in figure 4-12. Many requests of the same name may exist. To accommodate this, a linear list of requests must be maintained, this is the seventh

name	*name*	- String table entry containing the request name
materials	*materials*	- Dynlist of the materials' string table entries
results	*results*	- Dynlist of the results' string table entries
se	*se*	- Condlist of the external stimulation
ti	*ti*	- Condlist of the internal termination
te	*te*	- Condlist of the external termination
next	*next*	- Next request of the same name

Figure 4-11. Request Symbol Table Entry

component in figure 4-11.



allocated - The number of string table pointers allocated.

used - The number of string table pointers used.

dynptr - Pointer to the array of allocated pointers.

Figure 4-12. Dynlist Structure

There are two basic variants of data objects; dynamic longevity objects and the other (static and fluid) longevity objects. Figure 4-13 depicts the symbol table entry for static and fluid longevity objects. The value variant is populated for static longevity objects which are given an initial

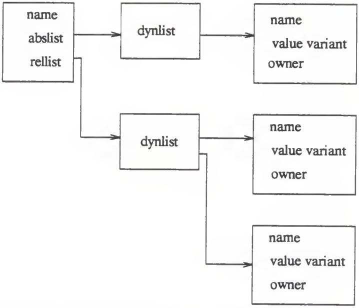
value in the object's declaration.

Figure 4-14 shows the layout for dynamic longevity objects. The initial entry is the "base" object built from the initial definition. This object owns lists of absolute and relative references to the object. These are dynamic longevity object symbol table entries as well but which have an integer value variant containing the absolute (or relative) reference number. The reference number is initially built in a sequence structure, shown in figure 4-15, and then propagated to the relative or absolute symbol table entry.

name	*name*	- String table entry containing the object
value variant	*value variant*	- Integer, real or character initial value

Figure 4-13. Symbol Table Entry for Static and Fluid Objects

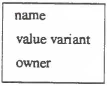
The remaining symbol table entry type is for labels. The label entry contains a pointer to the request's symbol table entry that it is associated with. Figure 4-16 describes this entry. Labels also present another problem for the compiler; they may be referenced in condition expressions prior to being parsed. To allow forward references of this type, a list of unresolved labels is maintained. This list, whose entries are shown in figure 4-17, is scanned after the parse has completed so that these references may be resolved.



Head symbol table element for a dynamic object:

name - String table entry containing the object's name
abslist - Dynlist of absolute references to this object
rellist - Dynlist of relative references to this object

Absolute and relative dynamic object (leaf) symbol table elements:



name - String table entry containing object's name
value variant - Relative or absolute instance
owner - Pointer back to head element

Figure 4-14. Symbol Table Structure for Dynamic Objects

type	*type*	- Relative or absolute reference
value	*value*	- The instance referenced

Figure 4-15. Dynamic Object Sequence Structure

name	*name*	- The string table entry for the label
requestname	*requestname*	- The symbol table entry for the labeled request

Figure 4-16. Symbol Table Entry for Labels

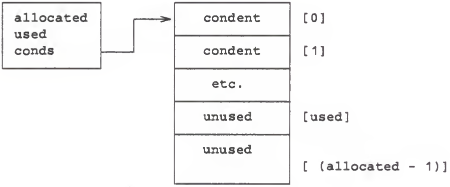
name	*name*	- The string table entry for the unresolved label
ref	*ref*	- The referencing condition entry
next	*next*	- The next entry on the unresolved label list

Figure 4-17. Unresolved Label List

4.3.5 Conditions Stimulation and termination conditions present an interesting problem; they must be evaluated, potentially many times, when the model is running. This implementation has chosen to translate the condition expressions into a list of condition entries in reverse polish notation. The virtual machine is viewed as capable of performing a stack based interpretation of this list.

The condition lists are rooted in a request symbol table entry. These lists are managed with the condlist structure shown in figure 4-18. The condition entries in the variable portion of the condlist are represented by the condent structure of figure 4-19.

The opcode in the condition entry is either "PUSH", a simple mathematical operator, a relational operator, or the special success or fail builtin functions. The operand within the condition entry structure is determined by the operand's type. Figure 4-20 describes the operand types and the value contained in the operand variant.



allocated - The size of the condition entry array.
used - The number of condition entries used in the array.
conds - Pointer to the variable sized array of condition entries.
condent - Condition entry (described in the next figure).

Figure 4-18. Condition List Management Structure

opcode	*opcode*	- The instruction to be executed
type	*type*	- The operand's type
operand	*operand*	- The operand as described below

The *operand* variant of the condition entry is:

integer value	*integer value*	- Immediate integer value
real value	*real value*	- Immediate real value
character value	*character value*	- Immediate character value
object pointer	*object pointer*	- String table pointer for data
request pointer	*request pointer*	- String table pointer for request

Figure 4-19. Condition Entry Structure

Operand Type	Operand's value
Integer	An immediate integer value is stored in the integer value element.
Real	An immediate real value is stored in the real value element.
Character	An immediate character value is stored in the character value element.
Fail/Success	The fail and success functions reference the symbol table entry for the labeled request through the request pointer element.
Object	The string table entry for the object is stored in the object pointer element.

Figure 4-20. Determination of the Operand Variant

4.3.6 Static Semantic Checks The following list enumerates the static semantic checks which are performed by the compiler. As mentioned previously, these are performed within the C language augmentations to the yacc parser specification.

— When initial values are specified they must match the type (int, char, or real) of the object being initialized.

— Initial values may only be specified for objects of static longevity.

— Objects referenced in material, results, or conditions must be defined.

— Objects referenced with an absolute or relative sequence number must have been defined with longevity dynamic.

— Relative references within conditions and material lists must not be positive.

— Relative references within results lists must be "..+1".

— Absolute references are not allowed in result lists.

— When an internal termination condition is specified, one and only one dynamic object must be contained in each of the material list, result list and the internal termination condition.

— All references within success and fail conditions must be to labeled requests. The references must be to a label.

— A string used to name an object, request or label cannot also name another object, request, or label.

4.3.7 Virtual Machine Output The output of the compiler is built for a virtual machine. This machine is viewed as a graph executor. In addition, as noted above, the machine has the capability of performing a stack based interpretation of the stimulation and termination conditions. This section will describe the graph the virtual machine executes. The condition section defined the format of the condition lists built and this section will not elaborate further.

Execution is driven by the receipt of "data events". These events correspond to the termination of a request and the subsequent availability of its results. The basic execution model of the virtual machine is specified in figure 4-21.

The basic execution graph necessary to drive this model has each object pointing to all of the requests which use the object as a material. The requests in turn point to all objects which are results of the request. This graph is represented in figure 4-22.

Two complications require extensions to the execution graph of figure 4-22:

1. Dynamic objects will cause the execution graph to form a network rather than a tree. This is because multiple relative instances (and absolute instances created under guarded requests) may be specified. This implies that the compiler cannot statically allocate all storage necessary for the graph's execution. The virtual machine is defined to be capable of dynamic storage allocation for dynamic longevity objects. The virtual machine must also be capable of managing the binding of relative names to the dynamically allocated storage.

2. When an available data object is found which is the material for an idle request, the availability of the request's other materials must be ascertained. To simplify this search, requests should point back to all objects in their material list.

The extended execution graph is presented in figure 4-23.

The symbol table forms the basis for the construction of the execution graph. Figure 4-24 depicts the symbol table's logical structure upon completion of the parse for a simple model. The symbol table easily maps into the execution model by simply building one more list linking each object with all of the requests which use it as a material. Figure 4-25 depicts this additional

Load the Program

While requests are in the idle or enabled state
do

- Scan all idle and enabled requests and evaluate each of their external termination conditions. Disable all of the requests in which their external termination evaluates to true.

- Scan the object list finding objects with existent and available values. For each of these:

    - Find all requests which use this object as a material.

    - For each of these requests in turn determine if all of its materials are available. If so, and the external stimulation condition evaluates to true, dispatch the request.

- Wait for the next data event.

- If the event is due to an iteration request

    - If the request failed, disable the request.

    - If the internal termination condition evaluates to false, redispatch the request. Do not rebind the materials.

    - If the internal termination condition evaluates to true, bind the results and disable the request.

- If the event is due to a loop request

    - If the request failed, disable the request.

    - If the external termination condition evaluates to true, bind the results and disable the request.

    - Otherwise, bind the results and return the request to the idle state.

- Otherwise,

    - If the request failed, disable the request.

    - If the request succeeded, bind the results and disable the request.

done

Figure 4-21. Virtual Machine's Execution Algorithm

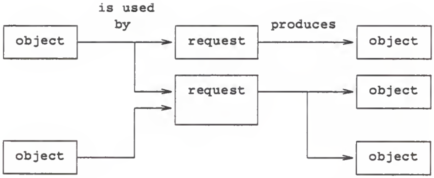


Figure 4-22. Basic Execution Graph

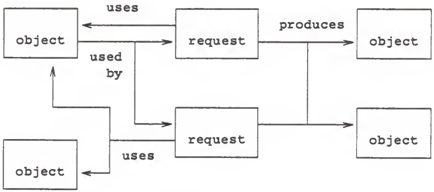


Figure 4-23. Extended Execution Graph

list.

The symbol table, with the additional list, is then dumped into an output file. This dumping process must map core pointers into symbolic pointers. This process will not be presented here.

Given the code:

R1(w, x; y)
R2(y; z)

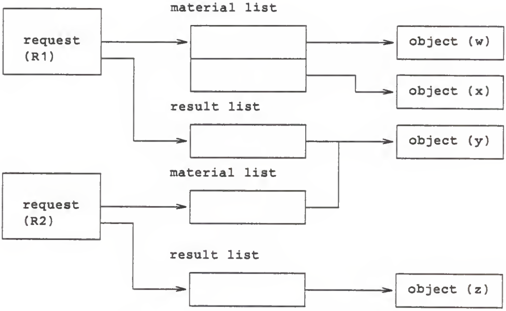


Figure 4-24. Symbol Table Graph after Parse

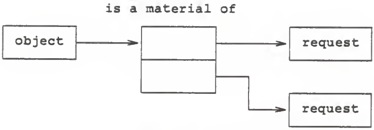


Figure 4-25. Object to Request Correspondence

5. Conclusions and Future Directions

5.1 Conclusions

The advent of multiple processor architectures requires a different approach to problem expression to fully utilize the available computational power. The primary problem is the partitioning of a problem and the ordering of the partitions. Explicit and implicit partitioning methods have been proposed. Data flow analysis is a particularly amenable method to detecting parallelism and forms the basis for implicit partitioning. The size of each partition is governed by the granularity of the underlying architecture. Ordering of partition execution is determined by the principle of data drive.

The ACM model forms the basis for this research. It is an intrinsically concurrent computational model. This model, with the augmentations presented, provides a very powerful and simple tool for describing concurrent processes. The model is abstract enough to conceal the underlying architecture from the user. This is beneficial in that the solution is not constrained by environmental restrictions.

The model's intrinsic approach to concurrency places the burden of synchronization, coordination, and communication in the realm of the systems software. This is also beneficial in that the user is freed to concentrate on the problem solution not the difficult tasks inherent in concurrent processing.

This thesis has made the following contributions to the study of the ACM model:

- Examined and reported experiences using the ACM model to solve a number of exercises.
- Perceived the necessity for, and defined, a loop request construct.

- Perceived the increased modeling power, and decreased complexity, attainable by introducing an implicit pipelined approach to objects of dynamic longevity. This capability was subsequently defined.
- Demonstrated the capability of modeling a complex, real world problem with the model and the proposed extensions.
- Specified the syntactic conventions and a compiler for a language based on a subset of the ACM model.
- Specified a virtual machine view of the compiler's output.

ACM has been found to be a very powerful tool for modeling concurrent processes. The augmentations provided in this thesis, as well as those in [Fis87] and [Yuk86], add to the model's power, flexibility, and ease of expression.

The majority of the examples presented in this thesis were at a macroscopic level. These exemplify the potential for developing a concurrent job control language based upon the model (such as one suggested in [Bra84]). There is also potential for using this model for the development of a module interconnection language ([DeR76]). These potential uses are in addition to the microscopic levels of concurrency which are the more traditional realm of dataflow models.

Although this thesis has advanced the study of ACM, it is but a minor step towards realization of a working and efficient product. The final section presents a list of open items which require attention.

5.2 Future Research Directions

The following list contains problems either discussed in this thesis, perceived but not discussed, or encountered during implementation of the specified system. These have been annotated in an effort to further "technology transfer" to future researchers.

- Physical Binding. Chapter two presented the notion that if a binding mechanism existed, simple types could be used to model very complicated underlying entities. For example, an atomic typed object might represent a physical device. An action might be bound to a process in the underlying machine, possibly even restricting this action to a particular processing entity. This sort of abstraction allows introduction of the model into existing environments. The delineation of a physical binding mechanism should be one of the next topics for investigation.

- Formal semantics. No attempt has been made to formalize the semantics of the model. [Fis87] provides some informal English descriptions of the static semantics of the specified language. This is a necessary, difficult and fruitful area for further investigation.

- Graphic interface. For many applications of this model, a textual syntax is an inappropriate vehicle for expressing the solution. The model's "language" could well be in a graphic format. Many of the examples in this thesis were presented in a graphical form recognizing that this was a better means of problem expression.

- Debugging. Debugging a data flow controlled program is non-trivial. The strengths of the approach (implicit concurrency, freedom from specifying scheduling information etc.) are exactly its downfalls when debugging. This could be a very interesting topic. If tied in with a "graphic language", an interactive debugger might be able to graphically display a request's

execution.

- Spatial coordinates. The problems of dynamic objects with spatial coordinates were presented in chapter two. These need to be investigated. There appears to be some debate over the subject of the ability of dataflow models to handle vectors and matricies within the computer science community. A potential starting place for investigation would be [Gaj82] and [Den83].

- Detailing. No effort was made to formalize the detailing concept. A few intellectual hurdles need to be overcome in this area:

  — Formal/actual parameters - do these exist? If so how? When does type checking occur? Is there a way or need to coerce types?

  — Instances of dynamic variables. At what point are these available to the external environment. Are all of the incarnations propagated? If not, which ones? Consider the prime number sequence described in chapter two, can this be done with any mechanism other than a loop request producing a pipeline of primes as shown in chapter three?

  — Hangup and Deadlock implications. What happens if one of these states is encountered? What if all of an action's results have been produced? What if only some have been produced? Is it possible to detect these states at compile time? If it is possible, what complexity (n log n, n squared etc.) is the algorithm which does this?

- Structures and partitions. The data encapsulation capabilities provided by structures and partitions have been acknowledged, but have not been examined. This should be a very interesting topic for further investigation. [Die85] discusses an approach taken towards partitions of data structures in an attempt to build a concurrent C language. This discussion

might be useful as a starting place for an implementation.

- FILE as a basic type. When the language was being specified, a basic type of FILE was investigated. This turned out to be a non-trivial problem. There appears to be little understanding of files above and beyond "streams" in the literature. (The model, as augmented in this thesis, will handle streams.) Adding a basic type of file would be very important. [Her84] discusses treating files as abstract data types, this might be a good starting point.

- Partialing. Partialing has also been acknowledged but not investigated. This subject becomes inter-mingled with detailing. The actual/formal parameter subject could be very interesting. The Multilisp [Hal85] concept of futures could be a good starting place for partialing research.

- Efficient Implementation of Dynamics. A key implementation detail is efficient management of dynamics. Database concepts of logging and replication might prove to be a good basis for implementing dynamics. [Her84] defines a "quorum" mechanism for obtaining up to date information in a distributed environment which might be an approach. Pipeline management is another issue. The compiler should be able to detect when no references are made to a dynamic object in the pipeline fashion. The compiler can then note this so that the runtime system need not maintain the queue.

- Scheduling. The scheduling algorithm is key for an efficient implementation. Locality, replications, and machine load would be balanced in determining where an action was dispatched. The implementation constructed as part of this research did none of these.

- Fault tolerance. If a multiple processor machine is used, processor failures must be expected.

In most cases the fault recovery capabilities of the underlying architecture will dictate the ability of the implementation to recover. Some further research is necessary to examine this in terms of a "virtual machine" to pinpoint the boundary between the implementation and the architecture.

- Introduction of Time. Stimulations and terminations could be based on time. The introduction of time (or any other external object) may render hangup and deadlock detection very difficult. These definitions within the original model should be re-examined when time is added.
- Support for libraries. Clearly a "real" development environment based on this model will require libraries and a separate compilation facility. Implementation of these lead to interesting questions in particular in the area of type checking. [Lis77] is a good starting place for an investigation into libraries.

BIBLIOGRAPHY

[Ack79] Ackerman, W.B., Data Flow Languages, Proceedings of the National Computer Conference, Volume 48, 1979, pages 1087-1095.

[Alm85] Almasi, G.S., Overview of Parallel Processing, Parallel Computing 2 (1985), pgs 191-203.

[And83] Andrews, G.R., and Schneider, F.B., Concepts and Notations for Concurrent Programming, Computing Surveys, Vol 15, No 1, March 1983, pgs 3-43.

[Arv80] Arvind, Kathail, V., and Pingali, K., A Data Flow Architecture with Tagged Tokens, Laboratory for Computer Science, Technical Memo 174.

[Bac78] Backus, J., Can Programming Be Liberated from the von-Neumann Style? A Functional Stytle and Its Algebra of Programs, Communications of the ACM, Vol. 21, No. 8, August 1978

[Bra84] Bracchi, G. and Pernici, B., The Design Requirements of Office Systems, ACM Transactions on Office Information Systems, Volume 2, No 2, April 1984, Pages 151-170.

[Bru83] Bruner, J.D., and Reeves, A.P., A Parallel P-Code for Parallel Pascal and Other High Level Languages, Proceedings of the 1983 International Conference on Parallel Processing, pages 240-243.

[Com78] Comte, D. *et. al*, Parallelism, Control, and Synchronization Expression in a Single Assignment Language, SIGPLAN Notices, January 1978.

[Cyt82] Cytron, R.G., Improved Compilation Methods for Multiprocessors, U of I Report No. UIUCDCS-R-82-1088, UILU-ENG 82 1713, May 1982.

[Dav81] Davies, J.R.B., Parallel Loop Constructs for Multiprocessors, U of I Report No. UIUCDCS-R-81-1070, UILU-ENG 81 1719, May 1981.

[Dav82] Davis, A.L., and Keller, R.M., Data Flow Program Graphs, IEEE Computer, Februrary 1982, pgs 26-41.

[Den80] Dennis, J.B., Dataflow Supercomputer, IEEE Computer, 13(11) (1980), pgs 48-56.

[Den83] Dennis, J.B., and Rong, G.G., Maximum Pipelining of Array Operations on Static Data Flow Machine, Proceedings of the 1983 International Conference on Parallel Processing, pages 331-334.

[DeR76] DeRemer, F., and Kron, H.H., Programming-in-the-Large Versus Programming-in-the-Small, IEEE Transactions on Software Engineering, Vol SE-2, No. 2, June 1976, pgs 80-86.

[Die85] Dietz, H., and Klappholz, D., Refined C: A Sequential Language for Parallel Programming, Proceedings of the 1985 International Conference on Parallel Processing, pgs 442-449.

[Dij68] Dijkstra, E.W., Cooperating Sequential Processes, In *Programming Languages*, F.Genuys, Ed., Academic Press, New York, 1968, pgs 43-112.

[Ens74] Enslow, P.H. (editor), *"Multiprocessors and Parallel Processing"*, John Wiley & Sons, Inc., 1974.

[Ezz85] Ezzat, A.K., and Agrawal, R., Making Oneself Known in a Distributed World, Proceedings of the 1985 International Conference on Parallel Processing, pgs 139-142.

[Fis87] Fish, R.W., Extensions to the ACM Dataflow Model, Masters Thesis, Kansas State University, 1987.

[Fly79] Flynn, M.J., Computer Organizations and Architecture, in *Operating Systems An Advanced Course", ed. Bayer, R. et. al.*, Springer-Verlag, 1979.

[Fra85a] Fraboul, C., and Hifdi, N., Expressing and Exploiting Parallelism on an Experimental MIMD System, Proceedings of the 1985 International Conference on Parallel Processing, pgs 236-238.

[Fra85b] Francez, N., and Yemini, S.A., Symmetric Intertask Communication, ACM Transactions on Programming Languages and Systems, Vol 7, No 4, October 1985, Pages 622-636.

[Fel79] Feldman, J.A., High Level Programming for Distributed Computing, Communications of the ACM, Volume 22, Number 6, June 1979, pgs 353-368.

[Gaj82] Gajski, D.D., Padua, D.A., Kuck, D.J., and Kuhn, R.H., A Second Opinion on Data Flow Machines and Languages, IEEE Computer, February 1982, pgs 58-69.

[Gaj83] Gajski, D., Kuck, D., Lawrie, D., and Sameh, A., Cedar - A Large Scale Multiprocessor, Proceedings of the 1983 International Conference on Parallel Processing, pages 524-529.

[Gaj85] Gajski, D.D., and Peir, J.K., Comparison of five multiprocessor systems, Parallel Computing 2 (1985) pgs 265-282.

[Gan75] Gannon, J.D., and Horning, J.J., Language Design for Programming Reliability, IEEE Transactions on Software Engineering, Vol SE-1, No. 2, June 1975, pgs 179-191.

[Geh84] Gehani, N., *"Ada Concurrent Programming"*, Prentice-Hall, 1984.

[Gel76] Gelly, O. *et. al.*, LAU System Software: A High Level Data Driven Language for Parallel Programming, Proceedings of the 1976 International Conference on Parallel Processing, pg. 255.

[Ghe85] Ghezzi, C. Concurrency in Programming Languages: A Survey, Parallel Computing 2 (1985) pgs 229-241.

[Gur85] Gurd, J.R., Kirkham, C.C. and Watson, I., The Manchester Prototype Dataflow Computer, CACM January 1985, Volume 28, Number 1.

[Hal85] Halstead, R.H., Multilisp: A Language for Concurrent Symbolic Computation, ACM Transactions on Programming Languages and Systems, Vol 7, No 4, October 1985, pages 501-538.

[Han73] Hansen, P.B., *Operating System Principles*, Prentice-Hall, Inc., 1973.

[Han75] Hansen, P.B., The Programming Language Concurrent Pascal, IEEE Transactions on Software Engineering, Volume SE-1, Number 2, June 1975, pages 199-207.

[Har85] Harris, J.P., Representing and Effecting Parallelism in Programs for Control-Flow Multiprocessors: Program Graphs and a Program Graph Interpreter, U of I Report No. UIUCDCS-R-85-1200, UILU-ENG-85-1705, January 1985.

[Her84] Herlihy, M.P., Replication Methods for Abstact Data Types, PH.D. Dissertation, June 1984, MIT.

[Hib78] Hibbard, P., Hisgen, A., and Rodeheffer, T., A Language Implementation Design for a Multiprocessor Computer System, Proceedings of the Fifth Annual Symposium on Computer Architecture, 1978, pages 356-362.

[Ho83] Ho, L.Y., and Irani, K.B., An Algorithm For Processor Allocation In A Dataflow Multiprocessing Environment, Proceedings of the 1983 International Conference on Parallel Processing, pages 338-340.

[Hoa72] Hoare, C.A.R., Towards a theory of parallel programming, in *Operating Systems Techniques*, Academic Press, New York, 1972, pgs 61-71.

[Hoa74] Hoare, C.A.R., Monitors: An operating system structuring concept, Communications of the ACM, Volume 17 Number 10, 1974, pgs. 549-557.

[Hoa78] Hoare, C.A.R., Communicating Sequential Processes, CACM August 1978, Volume 21, Number 8, pages 666-677.

[Jon80] Jones, A.K., and Schwarz, P., Experience Using Multiprocessor Systems - A Status Report, ACM Computing Surveys, Volume 12, No 2, June 1980, pages 121-165.

[Kuc74] Measurements of Parallelism in Ordinary FORTRAN Programs, Kuck, D.J. *et. al.* Computer, Volume 7, Number 1, January 1974.

[Kog85] Kogge, P.M., Function-based computing and parallelism: A review, Parallel Computing 2 (1985), pgs 243-253.

[Lel83] Leler, W., A Small, High-Speed Dataflow Processor, Proceedings of the 1983 International Conference on Parallel Processing, pages 341-343.

[Lie85] Liebowitz, B.H., and Carson, J.H., *"Multiple Processor Systems for Real-Time Applications"*, Prentice Hall, 1985.

[Lis77] Liskov, B., Snyder, A., Atkinson, R., and Schaffert, C., Abstraction Mechanisms in CLU, Communications of the ACM, Volume 20, Number 8, August 1977, pgs 564-576.

[Mot85] Motteler, H.E., and Smith, C.H., A Complexity Measure for Data Flow Models, International Journal of Computer and Information Sciences, Vol 14, No 2, 1985, pgs 107-122.

[McG82] McGraw, J.R., The VAL Language: Description and Analysis, ACM Transactions on Programming Languages and Systems, Vol 4, No 1, January 1982, pgs 44-82.

[Pad80] Padua, D.A., Kuck, D.J., and Lawrie, D.H., High Speed Multiprocessors and Compilation Techniques, IEEE Transactions on Computers, Vol. C-29, No. 9, Sept. 1980, pgs. 763-776.

[Par72] Parnas, D.L., On the Criteria to be used for Decomposing Systems into Modules, Communications of the ACM, Volume 15, Number 12, 1972, pgs 1053-1058.

[Per79] Perrott, R.H., A Language for Array and Vector Processors, ACM Transactions on Programming Languages and Systems, Volume 1, Number 2, October 1979, pages 177-195.

[Pet77] Peterson, J.L., Petri Nets, Computing Surveys, Vol 9, No 3, September 1977, pgs 223-252.

[Rei79] Reif, J.H., Data Flow Analysis of Communicating Processes, Conference Record of the 6th Annual ACM Symposium on Principles of Programming Languages, January 1979, pgs 257-268.

[Sri85] Srini, V.P., An Architecture for Doing Concurrent Systems Research, AFIPS Conference Proceedings, Volume 54, 1985 National Computer Conference, pages 267-277.

[Ung78a] Unger, E.A., A Natural Model for Concurrent Computation, Kansas State University Technical Report 78-35, Dissertation 1978.

[Ung78b] Unger, E.A., and Schweppe, E.J., A Concurrent Model: Basic Concepts, European Conference in Parallel and Distributed Processing, 1978.

[Wei71] Weinberg, G.M., *'The Psychology of Computer Programming*, New York: Van Nostrand Reinhold, 1971.

[Wil84] Williamson, R., and Horowitz, E., Concurrent Communication and Synchronization Mechanisms, Software Practice and Experience, February 1984, Volume 14(2), pages 135-151.

[Woo83] Woo, N.S., and Agrawala, A.A., The DC1 Flow Schema with the Data/Control-driven Evaluation, Proceedings of the 1983 International Conference on Parallel Processing, pages 244-251.

[Wul76] Wulf, W.A., London, R.L., and Shaw, M., Abstraction and Verification in Alphard, IEEE Transactions on Software Engineering, April 1976.

[Yuk86] Yuknavech, R.E., An Implementation of the ACM Dataflow Model, Masters Thesis, Kansas State University, 1986.

[Zar85] Zargham, M.R., and Purcell, R.D., A Protocol for Load Balancing on CSMA Networks, Proceedings of the 1985 International Conference on Parallel Processing, pgs 163-165.

APPENDIX 1 - SMART BNF GRAMMAR DESCRIPTION

This appendix presents the BNF grammar used as input to yacc to build the SMART parser. Terminals are presented in upper case and nonterminals are given in lower case.

```
pgm         :   varlist reqlist
            |   error

varlist     :   VAR dclist
            |   /* Null */

dclist      :   dclist dcitem SEMICOLON
            |   dcitem SEMICOLON

dcitem      :   longevity stype ID
            |   longevity stype ID LPAREN INTVAL RPAREN
            |   longevity stype ID EQ sign INTVAL
            |   longevity stype ID EQ sign REALVAL
            |   longevity stype ID EQ CHARVAL

longevity   :   FLUID
            |   DYNAMIC
            |   STATIC
            |   FIXED
            |   /* Null */

stype       :   INT
            |   REAL
            |   FILE
            |   CHAR

reqlist     :   reqlist request SEMICOLON
            |   request SEMICOLON

request     :   ID COLON rspec
            |   ID COLON se rspec te
            |   se rspec te
            |   rspec te

rspec       :   ID LPAREN mlist SEMICOLON rlist ti RPAREN
            |   ID LPAREN SEMICOLON rlist ti RPAREN
            |   ID LPAREN mlist SEMICOLON ti RPAREN
            |   ID LPAREN mlist ti RPAREN

se          :   condition
```

```
te          :   condition
            |   /* Null */

ti          :   condition
            |   /* Null */

mlist       :   mlist COMMA desig
            |   desig

rlist       :   rlist COMMA desig
            |   desig

desig       :   username instance

username    :   ID

instance    :   DOT DOT spatialpos DOT sequence
            |   DOT DOT spatialpos
            |   DOT DOT sequence
            |   /* Null */

spatialpos  :   LPAREN desig RPAREN
            |   LPAREN INTVAL RPAREN

sequence    :   INTVAL
            |   PLUS INTVAL
            |   MINUS INTVAL

condition   :   LBRACKET cond RBRACKET

cond        :   x bprime

bprime      :   OR x bprime
            |   /* Null */

x           :   y xprime

xprime      :   AND y xprime
            |   /* Null */

y           :   LPAREN cond RPAREN
            |   boolexp
            |   S LPAREN ID RPAREN
            |   F LPAREN ID RPAREN

boolexp     :   expr relop expr
```

```
expr     :  t eprime

eprime   :  PLUS t eprime
         |  MINUS t eprime
         |  /* Null */

t        :  f tprime

tprime   :  MULT f tprime
         |  DIV f tprime
         |  /* Null */

f        :  LPAREN expr RPAREN
         |  desig
         |  sign INTVAL
         |  sign REALVAL
         |  CHARVAL

relop    :  EQ
         |  NE
         |  GE
         |  GT
         |  LT
         |  LE

sign     :  PLUS
         |  MINUS
         |  /* Null */
```

A Programming Language and Compiler
Based on an Augmentation of
the ACM Model

by

Thomas Edgar Shirley

B. S., Carnegie-Mellon University, 1980

AN ABSTRACT OF A MASTER'S THESIS

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1987

ABSTRACT

Hardware architectures which contain multiple processing units are becoming practically available. Along with the increased processing power comes increased complexity. The difficulty of fully utilizing all processors, and coordinating the interactions of the programs running on them, cannot effectively remain under programmer control. Additionally, programmers should be shielded from the knowledge of the underlying architecture. This abstraction will promote more general and portable problem solutions, rather than solutions designed around a particular architecture.

A model for concurrent computation (ACM) has been developed ([Ung78a] and [Ung78b]) and is the starting point for this research. This model promotes the abstraction of data and procedure into objects. The procedural objects are driven by the availability of data objects and conditional stimulation and termination expressions. The model intrinsically provides concurrency.

In this thesis, the ACM model is briefly presented and discussed through a series of modeling exercises. The solutions modeled are pleasing in most cases, providing simple models for complex problems. A few exercises motivate the need for additions to the model; two are provided herein: support for an indefinite looping construct and a pipelined approach to data objects. A syntactic definition of a programming language for a subset of the ACM model is presented. A compiler for this language is specified as well as a virtual machine which executes the compiler's output.